

# Het effect van test driven development op de kwaliteit van software

S.J.M. Slagter

Master Software Engineering  
Universiteit van Amsterdam  
s.j.m.slagter@student.uva.nl

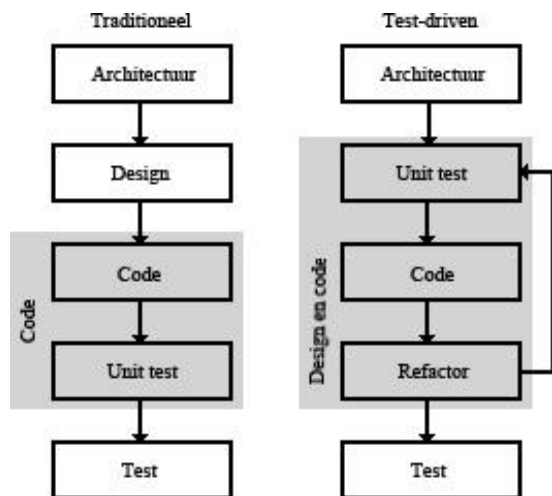
**ABSTRACT:** *Test Driven Development (TDD) is een relatief nieuwe benadering voor het ontwikkelen van software. Dit paper beschrijft het onderzoek naar het effect van deze nieuwe benadering op de kwaliteit van software. De informatie die voor dit onderzoek is gebruikt, is deels afkomstig uit de beschikbare literatuur en deels op basis van de bevindingen tijdens het Dancing Bear softwarehuis. Dit softwarehuis is een onderwijs project van de Universiteit van Amsterdam (UvA) wat is gericht op het leren en bestuderen van een software ontwikkel proces. De bevindingen uit dit softwarehuis en de bevindingen uit de literatuur zijn gebaseerd op het analyseren van code uit projecten van verschillende omvang, gemaakt in een traditioneel (test-last) ontwikkeltraject ten opzichte van code die is ontwikkeld in een test-first<sup>1</sup> ontwikkeltraject.*

**Keywords:** test-first, test-last, test driven development, TDD, software kwaliteit, interne kwaliteit, externe kwaliteit

## 1. Test Driven Development (TDD)

TDD is een techniek die gewenste verbeteringen of nieuwe functies in een software product toestaat op basis van korte iteraties waarin vooraf test cases

worden geschreven. Iedere iteratie produceert daarbij alleen code die benodigd is voor het laten slagen van de test. Een van de gedachten achter TDD is dat het de ontwikkelaars toestaat om snel te reageren op veranderingen in de code. Aanhangers van TDD beweren daarbij dat deze manier van ontwikkelen ervoor zorgt dat er minder code wordt geproduceerd die in de meeste gevallen ook nog eens eenvoudiger is om te begrijpen. Dit wordt gerealiseerd door kleinere klassen en routines en een minder sterke koppeling tussen objecten dan software die op basis van de traditionele software ontwikkelmethoden is ontwikkeld[1].



Afbeelding 1: Ontwikkelprocessen

<sup>1</sup> De termen Test-Driven en Test-First worden vaak verward. Test-Driven hoeft niet Test-First te zijn, andersom is dit wel het geval.

Een veel gehoorde misvatting over TDD is dat de methode zou gaan om het schrijven van geautomatiseerde testen of het schrijven van alle test scripts vooraf. Niets is echter minder waar. TDD is namelijk meer bedoeld als methode voor software design en niet slechts als methode om te testen[1,5].

Een groot voordeel van TDD is dat het de mogelijkheid geeft om software in kleine stapjes te ontwikkelen. Hierdoor kan er bijvoorbeeld sneller gereageerd worden op veranderingen maar ook het detecteren en verbeteren van problemen (bugs) kan vele malen sneller en gemakkelijker doordat er slechts in een klein stukje code gezocht hoeft te worden[6].

## 2. Software kwaliteit

Om het effect van TDD op softwarekwaliteit te kunnen bekijken is het belangrijk om eerst te weten wat software kwaliteit nu precies is en hoe je dit kunt meten. Om dit te kunnen doen verdelen we software kwaliteit over twee (hoofd)categorieën, namelijk; interne en externe software karakteristieken. Externe karakteristieken zijn daarbij voornamelijk interessant voor de gebruiker omdat deze hier direct mee te maken heeft, terwijl interne karakteristieken juist voornamelijk interessant zijn voor de ontwikkelaar[7]. Binnen de verdeling van interne en externe kwaliteitkarakteristieken kunnen we de volgende aspecten onderscheiden:

### Extern:

*Juistheid*  
*Bruikbaarheid*  
*Effectiviteit*  
*Betrouwbaarheid*  
*Integriteit*  
*Aanpassingsvermogen*  
*Nauwkeurigheid*  
*Robuustheid*

### Intern:

*Onderhoudbaarheid*  
*Flexibiliteit*  
*Portabiliteit*  
*Herbruikbaarheid*  
*Leesbaarheid*  
*Testbaarheid*  
*Begrijpelijkheid*

Vanwege het gebrek aan mogelijkheden om de externe kwaliteitsaspecten te toetsen heeft dit onderzoek zich alleen gericht op de interne kwaliteit van de software. De reden hiervoor is dat er binnen het Dancing Bear softwarehuis van de UvA aanzienlijk meer gegevens te verzamelen waren over de interne kwaliteitsaspecten dan de externe kwaliteitsaspecten.

Een andere reden waarom dit onderzoek zich slechts op de interne kwaliteitsaspecten heeft gericht is omdat de interne kwaliteitsaspecten meer benadrukken dat TDD veel meer een design methode is dan een test methode[1]. Dit uit zich voornamelijk in het feit dat het vooraf schrijven van testen de ontwikkelaar sneller inzicht geeft in de globale structuur van de code en eventuele wijzigingen hierin sneller en gemakkelijker zijn door te voeren.

### Meten:

Om nu daadwerkelijk het effect van TDD ten opzichte van deze interne kwaliteitsaspecten te kunnen meten, is er in dit onderzoek uitgegaan van een viertal belangrijke punten waarop de software is gecontroleerd. Op basis van deze punten is vervolgens vastgesteld hoe het met de kwaliteit van de software is gesteld na afloop van de ontwikkelfase. De hoofdpunten waarop de ontwikkelde software is gecontroleerd, zijn daarbij:

- 1) Error count,  
 Het aantal errors onderverdeelt in; critical, major, minor, annoyances en cosmetic.
- 2) Test coverage  
 Het percentage van gedekte code door testen.
- 3) Complexity  
 De complexiteit per methode of klasse op basis van het aan decision points.

#### 4) Comments

De hoeveelheid commentaar per onderdeel.

Uiteraard zijn er meer punten waarop gecontroleerd kan worden zoals coupling, accessors of de nested block depth. Dit zijn echter gegevens die zoals gezegd, op basis van de informatie beschikbaar uit het Dancing Bear softwarehuis, niet beschikbaar waren of waar weinig zinnige conclusies ten opzichte van TDD uit getrokken konden worden. Vandaar dat er voor de eigen bevindingen beperkt is tot eerdergenoemde punten.

### 3. Risico's

Voordat een onderzoek als dit van start kan gaan is het goed om eerst een aantal risico's te onderkennen. Zo is het mogelijk dat het onderzoek te maken krijgt met het zogenaamde Hawthorne effect[1]. Dit betekent dat het gevaar aanwezig is dat de deelnemers van het onderzoek, in dit geval de ontwikkelaars, zichzelf teveel gaan zien als een studiegroep en dus hun gedrag hierop gaan aanpassen[8]. Bijvoorbeeld door meer unit testen te schrijven dan dat ze normaal in dezelfde situatie zouden doen. Het spreekt voor zich dat dergelijk gedrag ervoor zorgt dat onderzoeksresultaten een stuk minder betrouwbaar worden.

Daarnaast kent het test driven ontwikkelproces op zichzelf ook een aantal risico's. Zo leeft er onder de meeste ontwikkelteams de angst dat een nieuwe methode als TDD de toch al strakke planning en naderende deadlines in gevaar zal brengen[2]. Daarnaast is het mogelijk dat ontwikkelaars voor het halen van de gewenste test coverage, testen schrijven die niet voldoen aan de gestelde eisen maar slechts 'domme' testen zijn ter aanvulling. Verder is het ook nog zo dat onervarenheid binnen het team kan zorgen voor een lagere test coverage.

Al deze punten zijn risico's die tijdens het onderzoek in de gaten zijn gehouden. In de praktijk blijkt namelijk dat deze risico's ervoor zorgen dat TDD slechts weinig of helemaal niet wordt gebruikt[2].

### 4. Literatuur

Zoals aangegeven heeft dit onderzoek zich voor een groot deel gericht op de literatuur. Daarom zullen nu eerst een aantal bevindingen hieruit worden besproken.

**Bron 1:** *Does Test-Driven Development Really Improve Software Quality [1]*

**Lines of code (Test-first):** 5.104

**Classes (Test-first):** 173

**Lines of code (Test-last):** 51.451

**Classes (Test-last):** 852

- 1) Test coverage is bij de test-last teams een stuk lager dan bij de teams die volgens test-first hebben gewerkt.
- 2) Ook het aantal lines of code die door (automatische) test suites werd gedekt was bij de test-last teams een stuk lager.
- 3) Test-first teams blijken kleinere stukken codes (modules) te schrijven dan de test-last tegenhangers en tevens kleinere methoden binnen deze modules.
- 4) De complexiteit, in termen van aftakkingen(branches) en het aantal methodes, van klassen is bij test-first teams een stuk lager dan bij de test-last teams.
- 5) De cyclomatic complexity, oftewel het mogelijk aantal executie paden door de code, was bij test-first teams lager dan bij de test-last teams.
- 6) Het aantal koppelingen tussen objecten was voor zowel de test-first als de test-last ongeveer gelijk. Helaas was deze ook in beide gevallen wat aan de hoge kant.
- 7) De test-first programmeurs blijken veelal methodes te schrijven met meer parameters om het testen van de methode te vergemakkelijken.

8) De test-first programmeurs blijken veel gebruik te maken van interfaces en abstracte klassen, ook om het testen te vergemakkelijken.

9) Test-first teams blijken meer abstractie in de code toe te voegen en meer sterk gekoppelde units te schrijven.

*Conclusie: test-first programmeurs schrijven software in meer kleinere units die minder complex zijn en beter getest.*

**Bron 2:** *Does writing software backwards really improve quality?[2]*

**Lines of code (Test-first):** Onbekend

**Lines of code (Test-last):** Onbekend

1) De ontwikkelde code op basis van TDD werd gedurende de ontwikkelfase vaker gerefactored.

2) Door middel van TDD was er meer vrijheid om te experimenteren, wat leidde tot betere design besluiten.

3) In een aantal gevallen bleek de ontwikkelaar geforceerd om gas terug te nemen en nog eens beter na te denken over hetgeen hij/zij probeerde te bewerkstelligen.

*Conclusie: TDD resulteert in minder defects in de code dan de traditionele ontwikkel methoden. De ontwikkeltijd nam echter wel met 15-20% toe maar waarschijnlijk neemt dit nog verder af naarmate de ontwikkelaars ervaring krijgen met het ontwikkelen op basis van TDD.*

**Bron 3:** *Software Quality Assurance and Testing [3,4,9]*

**Lines of code (Test-first):** 200

1) Code die via TDD is ontwikkeld slaagt in 18% meer functionele black-box test cases.

2) Het aantal defects in de code neemt eerst af met 62% en dan nog eens met 32% in twee opeenvolgende iteraties.

*Conclusie: De mate waarin TDD ondersteuning kan bieden in software ontwikkeling varieert. Het is echter wel duidelijk dat TDD ervoor zorgt dat er alleen testbare code wordt opgeleverd wat betekent dat de kosten van testen gereduceerd worden en uiteindelijk resulteert in betere kwaliteit van het product.*

**Bron 4:** *Driving Software Quality: How Test-Driven Development Impacts Software Quality [4]*

**Lines of code:** Onbekend

1) TDD produceert code die 18-50% meer externe test cases haalt dan niet test driven ontwikkel methoden.

2) TDD zorgt voor significante verbeteringen in de externe kwaliteitseisen.

3) TDD zorgt voor meer productiviteit

4) Code die is ontwikkeld aan de hand van een test driven ontwikkelmethode bevat veel minder unit-level bugs, minder functionele bugs en een significant hogere waarschijnlijkheid dat de verwachtingen van de stakeholders worden waargemaakt.

5) TDD is lastig om aan te leren.

*Conclusie: TDD zorgt voor een beter design van de te ontwikkelen software terwijl de complexiteit afneemt en de test coverage juist toeneemt. Daarnaast zorgt TDD ervoor dat business mensen en technische mensen beter met elkaar kunnen communiceren. Zelfs de ontwikkelaars kunnen door TDD onderling beter communiceren.*

## 5. Eigen bevindingen

Voor het onderzoek naar het effect van TDD op de kwaliteit van software is er gekeken naar de verschillende onderdelen van de "Graphbrowser". Deze Java applicatie is ontwikkeld als projectopdracht voor het Dancing Bear softwarehuis aan de UvA. De drie verschillende onderdelen die vallen onder deze Graphbrowser zijn:

1) de *exporter* omdat deze grotendeels test-driven is ontwikkeld en een relatief hoge test coverage heeft bereikt;

**Lines of code:** 206

**Classes:** 4

**Methods:** 26

**Decision points:** 45

2) de *layout* omdat dit één van de grootste onderdelen is binnen het project en niet test-driven is ontwikkeld;

**Lines of code:** 4.108

**Classes:** 50

**Methods:** 587

**Decision points:** 1.510

3) de *importer* omdat deze van vergelijkbaar formaat is met de exporter alleen daarentegen niet test-driven is ontwikkeld;

**Lines of code:** 196

**Classes:** 4

**Methods:** 27

**Decision points:** 61

Daarnaast is er ook gekeken naar het totale beeld van de Graphbrowser om de verkregen informatie mee te vergelijken. Hiervan zijn de gegevens als volgt:

**Lines of code:** 10.438

**Classes:** 189

**Methods:** 1.580

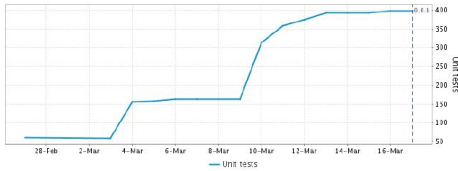
**Decision points:** 4.088

Helaas kunnen niet alle gegevens als even betrouwbaar worden beschouwd omdat is

gebleken dat er tijdens de ontwikkeling weinig draagvlak was voor het test-driven ontwikkelen. Dit is te wijten aan één van de eerder genoemde risico's, namelijk dat er ook in dit project volgens een strakke planning met harde deadlines werd gewerkt. Om te voorkomen dat de ontwikkeling achter kwam ten opzichte van de planning is er door de ontwikkelaars voor gekozen om op de oude vertrouwde manier door te ontwikkelen. Om toch enigszins te voldoen aan TDD is het exporter gedeelte wel grotendeels test-driven ontwikkeld maar helaas ook niet volledig. Mijnsinziens heeft dit ook te maken met een ander eerdergenoemd risico, namelijk onervarenheid met TDD binnen het ontwikkelteam. Deze onervarenheid in combinatie met de strakke planning heeft er dan ook voor gezorgd dat er binnen dit project nog weinig is geëxperimenteerd met TDD. In een toekomstig project kan hier wellicht rekening mee worden gehouden om de ontwikkelaars de kans te geven om te wennen aan TDD en dus de mogelijkheid te krijgen om een nieuwe techniek onder de knie te krijgen.

Ondanks deze vervelende ontwikkeling is er echter wel het een en ander te zeggen over TDD binnen het softwarehuis. Als er bijvoorbeeld wordt gekeken naar de importer en de exporter valt het op dat beide onderdelen beschikken over een hoge test coverage, namelijk 98.2% voor de exporter, welke test-driven is ontwikkeld en 98.8% voor de importer, welke op een traditionele methode is ontwikkeld. Hieruit valt op te maken dat TDD in dit project niet garant staat voor een hogere test coverage. Immers, de importer is op de traditionele wijze ontwikkeld en scoort desondanks een hogere test coverage. De layout heeft daarnaast een test coverage van 36.4% behaald.

Wel was het duidelijk zichtbaar dat het aantal testen in de loop van het project toenam (zie afbeelding 2).



Afbeelding 2: Aantal test in de tijd

Wat verder nog overeen komt met de bevindingen uit voorgaande onderzoeken is dat het test-driven onderdeel, in dit geval de exporter, voor een aanzienlijke lagere complexiteit zorgt. Dit is uiteraard een positieve ontwikkeling ten opzichte van de onderhoudbaarheid en flexibiliteit van de software. Hieronder volgt een klein overzicht:

Module / LOC	Layout / 4.108	Importer / 196	Exporter / 206
Methode	2.6	2.3	1.7
Klasse	30.2	15.2	11.2
Test Coverage	36.4%	98.8%	98.2%

Tabel 1: Complexiteit

De bovenste twee rijen met getallen in deze tabel wijzen op het aantal decision points per methode of klasse. Hierbij is te zien dat de complexiteit bij het test-first ontwikkelde onderdeel aanzienlijk lager ligt dan de test-last onderdelen. Aan deze tabel is verder ook direct op te maken dat wanneer de test coverage daalt, de complexiteit ook toeneemt. Een direct verband hiertussen is daarom aannemelijk alleen op basis van deze gegevens kan dit niet voor 100% worden gegarandeerd

omdat er meer factoren van invloed kunnen zijn op de complexiteit.

Een ander aspect waar iets over gezegd kan worden is de leesbaarheid en begrijpelijkheid van de code op basis van het commentaar in de code. Het blijkt dat code die via TDD is ontwikkeld aanzienlijk minder commentaar bevat dan test-last code. Dit is hoogstwaarschijnlijk te danken aan het feit dat de code met test-first een stuk minder complex is. De oorzaak van deze afgenomen complexiteit zit in de verdeling van de geschreven code over meerdere kleinere /simpelere methodes en/of klassen. Hierdoor spreekt de code meer voor zichzelf en hoeft er dus minder commentaar te worden toegevoegd. Dit duidt erop dat TDD in dat opzicht wel degelijk bijdraagt aan een betere kwaliteit van de software. Daarnaast komt het ook de testbaarheid ten goede omdat er meer testen per kleiner (minder complex) onderdeel geschreven kunnen worden maar eigenlijk is dat, gezien de gekozen test-first methode, een redelijk logisch resultaat.

Het aantal errors is in veel gevallen een goede indicatie of TDD nu echt een bijdrage levert aan betere software. Binnen het softwarehuis heeft dit echter geen toegevoegde waarde gehad. Wanneer namelijk de gegevens uit de volgende tabel (tabel 2) worden bekeken, dan valt het op dat er geen of weinig verschil kan worden waargenomen tussen test-driven en niet test-driven ontwikkelde code t.o.v. het aantal errors of failures. Het verschil tussen deze verschillende onderdelen is dusdanig klein dat er mijns inziens weinig conclusies uit getrokken kunnen worden. Hierbij mag uiteraard niet worden vergeten dat de layout module een stuk groter is (4108 lines of code) dan de overige twee modules (ongeveer 200 lines of code) en daarom dus ook meer fouten kan bevatten.

	Layout	Importer	Exporter
Errors	1	1	0
Failures	14	0	1

*Tabel 2: Errors en failures*

De conclusie die er dus getrokken kan worden ,naar aanleiding van de gegevens uit het Dancing Bear softwarehuis, is dat TDD wel degelijk zorgt voor minder complexe software die beter te begrijpen is dan test-last ontwikkelde software door het opdelen van de code in kleinere methoden en klassen. Daarnaast bevat de test-first code minder decision points en heeft het een hoge test coverage met een hoog aantal slagende tests (75%). Helaas kunnen er niet veel meer conclusies uit de gegevens gehaald worden omdat er te weinig gehouden is aan het test-first ontwikkelen. In een ideale situatie zouden er 2-3 onderdelen volledig test-first ontwikkeld moeten worden tegenover 2-3 onderdelen van vergelijkbare grootte die volledig test-last ontwikkeld zouden moeten worden. Gelukkig is er voldoende literatuur over het onderwerp en kan er uit andere onderzoeken (zie hoofdstuk 4) voldoende aanvullende informatie, zoals nested block depth (NBD), coupling, cyclomatic complexity, cohesion, fan-in/out en accessors gevonden worden. Deze informatie was helaas binnen het Dancing Bear softwarehuis niet beschikbaar. Dit heeft ervoor gezorgd dat de resultaten uit de eigen bevindingen wellicht een onvolledig of onjuist beeld geven om harde conclusies ten opzichte van TDD te trekken.

## 6. Conclusie

Wanneer we de informatie uit de literatuur en de informatie uit het eigen onderzoek samenvoegen dan blijkt dat er toch nog een duidelijk beeld over TDD naar voren komt.

Wat voornamelijk opvalt is dat TDD al snel zorgt voor kleinere 'brokken' code die op zichzelf al minder complex zijn en over het algemeen beter getest zijn dan de niet test-first ontwikkelde code. Mede hierdoor wordt code door TDD leesbaarder en beter te begrijpen en zorgt het voor een betere test coverage en mede daardoor voor minder problemen.

Desondanks denk ik dat TDD een methode is die niet voor ieder project toepasbaar zal zijn. Naar is gebleken in het Dancing Bear softwarehuis is het lastig om TDD toe te passen op projecten waarbij er weinig ervaring of weinig draagvlak is met TDD onder het ontwikkelteam. Wanneer er in zo'n geval toch voor TDD gekozen wordt, dan is de kans groot dat de ontwikkeltijd flink toeneemt en er slechts test worden geschreven die voldoen aan de criteria om de test succesvol af te ronden zonder dat deze de applicatie daadwerkelijk goed testen. Het is dus zaak om vooraf een goede afweging voor TDD te maken en eventueel de ontwikkelaars daarin op te leiden voorafgaand aan het project of om bewust de keuze te maken om het betreffende project tevens als een leertraject te beschouwen..

Bij een succesvol ontwikkeltraject op basis van TDD is onder andere uit de literatuur gebleken dat het wel degelijk zorgt voor een toenemende kwaliteit van de software in zowel een afname in het aantal fouten, een beter design, minder complexiteit en een hogere test coverage. Waarom TDD dan toch nog zo weinig wordt toegepast is waarschijnlijk meer een psychologische dan een technische kwestie en zeker de moeite waard om te onderzoeken. Helaas valt dit buiten de scope van dit onderzoek.

## Bronnen

[1] David S. Janzen and Hossein Saiedian *Does Test-Driven Development Really Improve Software Quality*, 2008

[2] Grant Lammi *Does writing software backwards really improve quality?*

[3] Ben Kim *Software Quality Assurance and Testing*, 2008

[4] Lisa Crispin *Driving Software Quality: How Test-Driven Development Impacts Software Quality*, 2006

[5] Wikipedia: Test-driven-development, [http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)

[6] Agiledata: Parting Thoughts, <http://www.agiledata.org/essays/tdd.html#PartingThoughts>

[7] Steve McConnell *Code Complete*, 2004

[8] Wikipedia: Hawthorne effect, [http://en.wikipedia.org/wiki/Hawthorne\\_effect](http://en.wikipedia.org/wiki/Hawthorne_effect)

[9] George, B and Williams, Laurie A *Structured Experiment of Test-Driven Development*, 2004