Test Driven Development vs. Non Test Driven Development

Mads Andersen
Benjamin Højgaard

Maj 2021

1 Introduktion

I denne artikel har vi gennemgået forskellige Test Driven Development (TDD) undersøgelser og statistikker for at forstå hvordan det bliver brugt i praksis, fordelende og de udfordringer TDD bringer for et udviklingsteam.

Traditionelt har softwareudvikling været en lineær proces, men gennem de sidste årtier er det blevet mere populært at udvikle agilt, op imod 87% [8] af udviklingteams følger en agil eller agil-lignede tilgang.

Softwareudvikling er begyndt at omfavne forskellige metoder, der tager højde for projektets krav og karakter, Test Driven Development er en af de metoder som har tiltrukket sig opmærksomhed i softwareudviklingsmiljøet. I en bogen "Test Driven Development: by example.", skrevet af den anerkendte softwareingeniør Kent Beck, giver han også udtryk for at Test Driven Development er en af hjørnestenene i Extreme Programming (XP) udviklingsmetoden [2].

Selv siger den selvudråbte genopdager af TDD [7] Kent Beck, *"test-driven development (TDD), a proven set of techniques that encourage simple designs and test suites that inspire confidence."* [2], men vi står stadig tilbage med spørgsmål om kvalitets- og produktivitetskravne der bliver sat i forbindelse med TDD.

Denne artikel starter med at definere begrebet TDD og hvordan det adskiller sig fra den traditionelle unittest tilgang. Derefter kigger vi på nogle af de statistikker, der enten validerer eller afviser påstandene om TDD.

2 Hvad er Test Driven Development?

Test Driven Development er en tilgang, hvor en test skrives, inden softwareudvikleren opretter produktionskoden for at udføre testen. Denne tekniks grundlæggende idé er at give kodeforfatteren tid til at overveje deres design eller krav, før han/hun skriver funktionel kode.

2.1 Test Driven Development Processen

Test Driven Development kan deles op i forskellige processer

1. **Skriv testen:** I TDD begynder alle nye features med at skrive en test. Udvikleren skal forstå de specifikationer og krav featuren skal opfylde. For at opnå dette, bliver de nød til at gennemgå user stories og/eller use cases for at forstå formålet med den nye kode de skal til at udvikle.

2. **Testen fejler:** Når programmøren har lavet testen, kører han/hun den. Da implementationen af featuren endnu ikke er udviklet, vil testen fejle.

3. **Skriv koden:** Nu ved programmøren at featuren virker hensigten. Programmøren skal nu skrive den mindst muglige implementation så testen består, det betyder nødvendigvis ikke at koden er perfekt, men det er lige meget for udvikleren forventes ikke at lave funktionalitet udover det testen tjekker for.

4. **Refactor kode:** Når kodebasen vokser skal den vedligeholdes og rengøres kontinuerligt. I de overstående faser var programmørens hovedfokus kun at skrive kode, skal denne fase sikrer effektivitet. Det muliggør at programmets kildekodes interne struktur bliver forbedret, mens dets eksterne egenskaber bevares.

5. **Gentag:** Ovenstående trin gentages automatisk for at sikre at TDD cyklussen dækker alle funktionerne.

3 Forskelle mellem TDD og traditionel udvikling

For at forstå TDD virker det hensigtsmæssigt at se hvordan det adskiller sig fra de traditionelle tilgange til programmering.

Hovedforskellen er at TDD kører i en cyklisk proces, hvor de traditionelle metoder følger en lineær proces.

Programmører som udvikler de traditionelle testmetoder starter med at skrive sin kode og koncentrerer sig kun om testen i slutningen af udviklingsprocessen. I modsætning til TDD, hvor man starter med at skrive testen inden man udvikler sin kode der opfylder testen.

Programmøren der bruger den traditionelle tilgang kan være meget opmærksom på kodens kvalitet, men risikerer at de ikke opdager alle kodens fejl, hvorimod programmøren der bruger TDD arbejder på koden indtil testen består. Dette gøres indtil koden opfylder funktionaliteten, hvilket sandsynligvis vil resultere i færre fejl.

4 Fordele og ulemper

Vi vil i de næste par afsnit se på nogle cases hvor TDD blev anvendt, og hvilke resultater der kom ud af disse cases.

4.1 Kvalitet kontra tid

En bekymring man kan have, når man skal overveje at benytte TDD til fordel for anden udviklingsmetode kan være at det forlænger udviklingstiden for et givent projekt.

En undersøgelse der involverede teams fra IBM og Microsoft konkluderede at disse teams oplevede en forlænget udviklingstid på mellem 15-35% når de brugte TDD. Undersøgelsen understreger dog, at denne vurdering blev subjektivt estimeret af ledelsen [5].

Til gengæld mente disse teams at maintenance omkostningerne ville reduceres grundet den øgede kvalitet som TDD medførte [5].

En anden undersøgelse, hvor programmørerne ikke var klar over deres deltagelse i en undersøgelse, siger *"While the development of both the systems utilizing TDD took extra time upfront the resulting quality was higher than teams that adopted a non-TDD approach by an order of at least two times."*[6].

Men igen pointeres det, at kvaliteten af det stykke software der blev lavet var højere, og mængden af defekter var lavere.

En yderligere undersøgelse hvor der blev udført et struktureret experiment, der involverede 24 pair-programmører, hvor 12 brugte TDD, og 12 brugte en non-TDD tilgang. Resultatet af denne undersøgelse viste, at TDD-programmørerne brugte længere tid, men softwaren bestod flere black-box tests, *"... indicate that TDD programmers produce higher quality code because they passed 18% more functional black-box test cases. However, the TDD programmers took 16% more time."* [3]

De førnævnte undersøgelse understøtter alle den samme teori, at TDD er med til at øge kvaliteten af software, mens det kan forlænge udviklingstiden. Dog bør det påpeges, at disse undersøgelser alle har en relativt lav sample size, og kulturene i de givne virksomheder er forskellige, så resulaterne antyder at TDD til fordel kan benyttes i stedet for en non-TDD tilgang.

4.2 Defekt-reduktion med TDD

En af de påståede fordele ved at benytte TDD er, at det stykke software der er under udvikling har færre fejl. I en undersøgelse foretaget af Nagappan et al, målte man på hvor stor en *defect density* de stykker software som deltagerne af undersøgelsen udviklede havde. Nagappan et al definerer defect density som *"defects/thousand lines of code (KLOC)"*[5], hvor en defekt defineres som *"When software is being developed, a person makes an error that results in a physical fault (or defect) in a software element. When this element is executed, traversal of the fault or defect may put the element (or system) into an erroneous state. When this erroneous state results in an externally visible anomaly, we say that a failure has occurred".*[1]

Deltagerne var fire teams fra forholdsvis IBM og Microsoft, et fra IBM, og tre fra Microsoft. Alle teams brugte TDD, og resultaterne af undersøgelsen er sat op imod forrige projekter i de respektive virksomheder, som benyttede en non-TDD tilgang.

Alle teams oplevede en væsentlig nedgang i defect density. IBM oplevede en nedgang på 40%, mens holdene fra Microsoft oplevede en nedgang på 60-90%.[5]

I en tilsvarende undersøgelse, der blev gennemført hos et team fra en anden afdeling i IBM, oplevede man også en nedgang i defect density på 50% [4]. Igen blev resultatet sammenlignet med tidligere lignende projekter, der benyttede sig af en non-TDD tilgang.

Resultaterne fra de ovenstående undersøgelser tyder på, at TDD kan have en stor effect på antallet af fejl softwaren besidder. Eftersom undersøgelserne kun strækker sig over fem teams, er det svært at komme med en endelig konklusion.

5 Konklusion og fremtidigt arbejde

Test Driven Development vister sig at have fordele og ulemper. Tendensen fra de undersøgelser vi har set på, er at kvaliteten af software hæves markant, med en defekt reduktion på op til 90%, samt lavere maintenance omkostninger, samtidigt bliver udviklingstiden forhøjet med 15 til 35%.

Yderligere kontrolleret undersøgelser på en større skala i industrien og i den akademiske verden, kan validerer eller modbevise disse resultater.

References

[1] IEEE Computer Society. *IEEE guide for the use of IEEE standard dictionary of measures to produce reliable software.* 1988.

[2] Kent Beck. *Test-driven development: by example.* Addison Wesley, 2002. ISBN: 0-321-14653-0.

[3] Boby George et al. *A structured experiment of test-driven development.* URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.147&rep=rep1&type=pdf.

[4] E. Michael Maximilien et al. *Assessing Test-Driven Development at IBM.* URL: https://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF.

[5] Nachiappan Nagappan et al. *Realizing quality improvement through test driven development: results and experiences of four industrial teams.* URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2009/10/Realizing-Quality-Improvement-Through-Test-Driven-Development-Results-and-Experiences-of-Four-Industrial-Teams-nagappan_tdd.pdf.

[6] Thirumalesh Bhat et al. *Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies.* URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.187.9671&rep=rep1&type=pdf.

[7] Kent Beck. *Why does Kent Beck refer to the rediscovery of test-driven development? What's the history of test-driven development before Kent Beck's rediscovery?* URL: https://www.quora.com/Why-does-Kent-Beck-refer-to-the-rediscovery-of-test-driven-development-Whats-the-history-of-test-driven-development-before-Kent-Becks-rediscovery.

[8] Lalit and Joel. *State of Testing 2019 Annual Report.* URL: https://www.practitest.com/assets/pdf/STOT_ver_1_3.pdf.