Waarom waterval niet werkt

@TheTweetOfGod: To understand the universe you need to see it in the broader context.

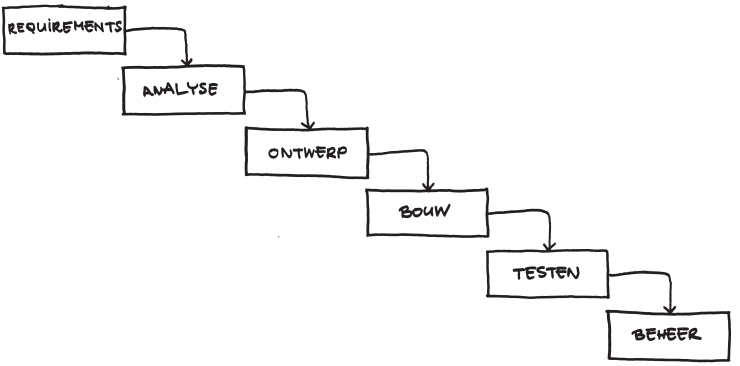
In de jaren zeventig van de vorige eeuw was de Amerikaan Winston Royce directeur van Lockheed's Software Technology Center. Hij beschreef als eerste het model dat iedereen nu kent als waterval. Hij deed dat in 1970 in zijn whitepaper *Managing the Development of Large Software Systems*. Hoewel dit paper als de moeder van alle watervalmethodieken wordt gezien, toont Royce zich geen voorstander van dit model. Sterker nog. Royce gebruikt het watervalmodel als voorbeeld van een proces dat *niet* goed werkt.

De kenmerken van waterval

Het watervalmodel is een sequentieel proces waarin alle werkzaamheden om software te realiseren na elkaar worden uitgevoerd. Royce geeft deze werkzaamheden als volgt weer.

Royce beschouwt deze werkzaamheden als de minimaal noodzakelijke om succesvol software te realiseren en in onderhoud te nemen. Nog altijd zijn veel organisaties hiërarchisch ingedeeld volgens dit model. Er is een afdeling voor requirements of business analyse. Een afdeling voor informatie-analyse. Een voor functioneel en technisch ontwerp. Er zijn ontwikkelaars. Er zijn testers. Er is een afdeling met functioneel en technisch beheerders. En natuurlijk zijn er de onvermijdelijke projectmanagers die de andere rollen in het gareel moeten houden.

Hoewel Royce de term zelf niet gebruikt, dankt ons vakgebied het watervalmodel direct aan deze afbeelding uit 1970.



Het watervalmodel kenmerkt zich als volgt:

- **Fasen.** Ieder project telt de vaste fasen zoals die in de afbeelding zijn weergegeven.
- **Eenmalig.** Ieder van deze fasen wordt eenmalig en in deze volgorde uitgevoerd.
- **Rollen.** Iedere fase kent zijn eigen specifieke rol. Analisten doen analyse, ontwikkelaars ontwikkelen en testers testen.
- **Eindproducten.** Aan het eind van een fase wordt een eindproduct opgeleverd. Signed, sealed and delivered. Het zijn de zogenaamde *milestones*. *De* informatie-analyse of *het* functioneel ontwerp.
- **Volledig.** Een volgende fase start pas als het eindproduct uit de vorige fase volledig en compleet is opgeleverd. Het bouwen van de software start pas als het ontwerp gereed is. En pas als de software ontwikkeld is, wordt er getest.
- **Vast.** Projecten worden in de regel uitgevoerd met een van tevoren vastgesteld budget, een vaste einddatum en een vaste scope.

Klinkt logisch toch?

De problemen met waterval

Sinds 1970 is het leeuwendeel van alle projecten uitgevoerd volgens (variaties op) het watervalmodel. Een belangrijk deel van deze projecten was niet succesvol. Vrijwel iedereen in ons vakgebied herinnert zich wel projecten die ver over budget gingen, de afgesproken deadlines niet haalden, slechts een deel van de afgesproken requirements implementeerden of vroegtijdig werden gestopt, al dan niet na diverse herstarts.

Kijkend naar de kenmerken van deze *traditionele* projecten zijn hiervoor heel duidelijke redenen aan te wijzen. Ik noem de meest voorkomende:

- **Kennis verdwijnt.** Iedere fase wordt uitgevoerd door één specifieke rol. Zodra het eindproduct van een fase is opgeleverd, verdwijnt deze rol uit het project. Hiermee verdwijnt ook na iedere fase veel kennis uit het project. Immers, hoe veel iemand ook documenteert, het is onmogelijk om *alles* te documenteren.
- **Voortschrijdend inzicht.** Gedurende iedere fase van een project leert het team meer over de scope van het project, de wensen van de gebruikers en de (on)mogelijkheden van de gekozen technologie. Dit heet *voortschrijdend inzicht*. Maar omdat in de eindproducten van eerdere fasen alles al vastligt, kan dit inzicht niet meer worden meegenomen.
- **Wijzigende requirements.** Gemiddeld wijzigen de requirements gedurende een project 20 tot 25 procent. Er worden nieuwe requirements gevonden. Bestaande requirements wijzigen of komen te vervallen. Maar omdat alle requirements al in een vroegtijdig stadium tot in detail zijn uitgewerkt, blijkt een deel van dit werk overbodig of moet opnieuw worden uitgevoerd. Dit is inefficiënt en duur.

  Een projectmanager vertelde mij ooit: 'Waterval werkt prima. Zolang je maar geen wijzigingen toestaat.'

- **Compleet en volledig.** Omdat het eindproduct van een afgeronde fase niet meer wijzigt, is het zaak 100 procent compleet

en volledig te zijn. Vooral tijdens analyse en ontwerp. Immers, als tijdens een volgende fase nieuwe requirements worden ontdekt, gelden deze als onvolkomenheden van eerdere fasen.

Een ontwerper: 'Als we meer tijd aan de analyse hadden besteed, waren we nu niet tegen deze problemen aangelopen.'

Hierdoor vinden met name analisten en ontwerpers het moeilijk hun eindproducten definitief op te leveren. Vroege fasen van projecten lopen hierdoor vaak ernstig uit.

- **Overdocumentatie.** Soms wordt er om compleet en volledig te zijn zodanig veel gedocumenteerd dat het onmogelijk is om ooit nog de software te bouwen, laat staan te testen.

  In een project bij een bank in België was twee jaar besteed aan de functionele analyse. Er waren ruim 2500 pagina's tekst geproduceerd. De beoogde ontwikkelaars vertelden me dat ze niet eens een schatting van de omvang van het werk durfden te maken, laat staan eraan te beginnen.

- **Lastig schatten.** Iedere fase herbergt een ander type werkzaamheden en kent dus zijn eigen tempo. Hierdoor is het moeilijk goede schattingen af te geven. Want als de analyse zes maanden duurt, wat zegt dit dan over de totale doorlooptijd van een project?
- **Late risico's.** In waterval vindt het daadwerkelijk ontwikkelen van de software pas laat in het project plaats. Soms zijn er al enkele jaren verstreken voor de eerste code wordt geschreven. Dit brengt grote risico's met zich mee. De beoogde technologie kan in de tussentijd verouderd zijn. Ook start het testen van de software pas, nadat de code volledig is opgeleverd. Hoe goed de analisten, ontwerpers en ontwikkelaars ook zijn, testers vinden altijd onvolkomenheden. Doordat met het verstrijken van de fasen steeds meer werk is verricht, nemen de kosten voor het oplossen van fouten in een project exponentieel toe. Dit staat bekend als de Wet van Boehm.

Een mooi voorbeeld is een project van een grote internationale bank. Deze bank ontwikkelde nieuwe software voor het afhandelen van polisaanvragen. De architect had een prachtige architectuur bedacht waarbij validaties dynamisch werden afgevuurd op het moment dat de gebruiker naar een volgend veld ging. Het project vorderde gestaag. Na anderhalf jaar droegen de ontwikkelaars de software over aan de testers.

De ontwikkelaars beschikten echter over zware ontwikkelmachines, terwijl de testers dezelfde machines gebruikten als de eindgebruikers. Al op de eerste dag waarop werd getest, bleek de performance van de software belabberd. Alleen al de wijzigingen aan de architectuur die nodig waren om de performance te verbeteren, zorgden ervoor dat het project een halfjaar uitliep.

Waarom bestaat waterval nog steeds?

Een interessante vraag is natuurlijk waarom het watervalmodel nog steeds voorkomt. Allereerst is het een eenvoudig model, dat gemakkelijk uit te leggen is. Bovendien geeft het de illusie voorspelbaar en meetbaar te zijn. Niet zelden zien projectmanagers in het model een Gantt-chart met de planning van hun project. Ook is de mens van nature conservatief. We houden lang vast aan wat we eerder hebben gedaan – succesvol of niet. 'Ik werk al twintig jaar zo, waarom zou ik dat nu veranderen?' Projecten lijken na afloop vaak succesvol, omdat het eindresultaat niet meer wordt vergeleken met de oorspronkelijke doelstellingen en begroting. 'We zijn al blij dat we überhaupt opleveren.'

In feite is het watervalmodel een kopie van vergelijkbare modellen uit andere industrieën, die voor *herhaalde* productie worden aangewend. Denk maar aan de automobielindustrie. Software development laat zich echter slecht vergelijken met herhaalde productie. Anders dan bij de productie van steeds hetzelfde model auto, vergt software development een grote mate van creativiteit. Voortdurend worden beslissingen genomen die de uitkomst beïnvloeden, ook tijdens het ontwikkelen en testen van de software. De productiemetafoor gaat niet op. Een betere metafoor uit dergelijke

industrieën zou die van het ontwikkelen van een nieuw type auto zijn.

Deze nadelen van waterval zijn voldoende om te concluderen dat het model niet werkt en ook nooit heeft gewerkt. Talrijke onderzoeken bewijzen deze conclusie. Projectmanagers die aangeven dat hun watervalprojecten wel goed gaan, zijn de spreekwoordelijke uitzondering op de regel. Zij vergelijken de uitkomst van het project niet met de oorspronkelijke begroting of hebben hun werkwijze bijgesteld om deze nadelen te minimaliseren.

Van waterval naar iteratief

Opvallend genoeg kwam Winston Royce in 1970 al tot de conclusie dat waterval niet werkt. In zijn paper gebruikt hij dit model om te laten zien hoe het *niet* moet. Royce is voorstander van een *do-it-twice-approach*. Daarbij wordt de software in kleine delen geanalyseerd, ontworpen, ontwikkeld, getest en opgeleverd. Het is voortdurend toegestaan de feedback uit ieder van de werkzaamheden te gebruiken om eerdere milestones te verbeteren. Dus als tijdens het ontwikkelen blijkt dat het ontwerp moet worden verbeterd, dan gebeurt dit ook. Direct.

Eigenlijk predikt Royce veel eerder een *iteratief* model. Dit is opvallend, omdat zijn paper wordt gezien als de moeder van alle watervalmethodieken. Interessanter nog is dat we de ideeën van Royce na jaren van evolutie terugvinden in de huidige generatie aanpakken die stuk voor stuk een hoog iteratief karakter kennen. Dit is agile.

Wat is agile?

@Quote_Soup: Never be afraid to try something new.
Remember, amateurs built the ark. Professionals built the Titanic.

Vrijwel ieder boek over agile begint met het *Agile Manifesto*. Zo ook dit boek. Dit manifest biedt uitstekende uitgangspunten over agile. Het is tijdens een weekend in Utah in 2001 opgesteld door een groep mensen die hun sporen hadden verdiend in het ontdekken van vernieuwende aanpakken voor software development. Veel van deze aanpakken wezen in eenzelfde richting. Weg van de zware procesgeoriënteerde waterval-ontwikkelmethodieken.

Hoewel de groepsleden ook grote verschillen in inzicht hadden, legden ze vooral de nadruk op de gemeenschappelijke waarden van hun aanpakken. Ieder ondervond dat er nieuwe manieren van samenwerken aan het ontstaan waren. Deze werkten aantoonbaar beter dan die van traditionele methodieken. Kort samengevat omvat het manifest vier krachtige statements die nauwgezet de gemeenschappelijke waarden van deze nieuwe generatie aanpakken aangeven:

- Mensen en interactie over processen en tools.
- Werkende software over allesomvattende documentatie.
- Samenwerken met de klant over contractonderhandelingen.
- Inspelen op veranderingen over het volgen van een plan.

Elk van deze statements bevat het verbindingswoord *over*. Dit geeft aan dat de rechterkant van de statements belangrijk is, maar dat de linkerkant *nog* belangrijker is. En niet zoals vaak ten onrechte

wordt geïnterpreteerd dat de rechterkant verboden is. 'Nee. In agile documenteren we niet.' 'Nee. We schrijven geen plan. Hoezo?' De rechterkant van de vier statements is eveneens belangrijk. Ook in agile.

Aanpakken en methodieken

Natuurlijk kan het geen kwaad een werkwijze te volgen in projecten. Maar het tot drie cijfers achter de komma uitvoeren van een compleet uitgeschreven aanpak of methodiek is vaak te star. Dan liever lichtgewicht.

MENSEN EN INTERACTIE
OVER PROCESSEN EN TOOLS

Het Agile Manifesto benadrukt dat het vooral mensen in teams zijn en de manier waarop deze mensen samenwerken waardoor projecten succesvol zijn. Agile aanpakken bieden dan ook talrijke technieken om samenwerking vorm te geven. Vooral tussen de verschillende rollen in projecten.

Wat is eigenlijk het verschil tussen *aanpakken* en *methodieken*? Kortweg beschrijft een aanpak een proces of werkwijze zonder concreet alle werkzaamheden, rollen en producten te definiëren. Aanpakken zijn hierdoor breed toepasbaar. Denk aan Scrum, Kanban of Extreme Programming, maar ook aan PRINCE2. Scrum afficheert zichzelf zelfs vooral als raamwerk. Het hanteren van een aanpak of raamwerk geeft een project vrijheden. Soms laat dit projecten echter in verwarring.

Een methodiek beschrijft eveneens een proces of werkwijze, maar geeft daarbij concretere richtlijnen en best practices over werkzaamheden, technieken, rollen en producten. Methodieken zijn daardoor minder breed toepasbaar dan aanpakken, maar bieden wel meer houvast. En soms een beetje te veel. Denk aan SDM of Rational Unified Process (RUP).

Kenmerkend voor agile is dat projecten zich niet vastbijten in een bepaalde aanpak of methodiek, maar voortdurend hun werkwijze optimaliseren. En omdat ieder project anders is, verschilt ook

de werkwijze per project. Dit ondanks de fundamentalistische wind die door diverse agile gemeenschappen waait, waarin wordt aangegeven wat precies wel en niet mag. 'Modelleren mag niet in Scrum.' Of: 'In Kanban gebruik je altijd work-in-progress limits.'

Dit fundamentalisme is vooral zichtbaar bij populaire agile aanpakken. Daar is de instroom zo groot dat de spoeling soms dun wordt. Houd in het achterhoofd dat mensen en hun interactie altijd belangrijker zijn dan processen en tools. Juist in agile!

Werkende software

Veel mensen in traditionele projecten produceren vooral papier. Sterker nog, analisten en ontwerpers hebben het project al verlaten voordat er code en tests worden geschreven. Als analist of ontwerper zie je zo niet eens het resultaat van je noeste arbeid. In agile is dit anders.

WERKENDE SOFTWARE
OVER ALLESOMVATTENDE DOCUMENTATIE

In agile werken alle rollen samen. Ook de analisten en ontwerpers. Deze rollen worden in agile meestal beschouwd als *domeinexperts*. Maar ook de klant en de gebruikers werken samen. Met elkaar. Niet na elkaar. Tegelijkertijd. Iedereen ziet zo direct het resultaat van zijn werk.

Belangrijk is dan ook dat teams een optimale manier vinden om te documenteren. Het is dus niet zo dat agile teams *niet* documenteren. Agile teams documenteren net genoeg. Genoeg om de architectuur vast te stellen. Genoeg om te kunnen testen. Maar ook genoeg om de software na afloop van een project in beheer te nemen. Agile projecten proberen daartoe de minimale set aan documentatie te vinden om hun doelstellingen te bereiken. Het is verstandig om de uiteindelijke beheerders van de software bij het project te betrekken. Ook beheerders zijn stakeholder in het project.

Samenwerking en contracten

In traditionele projecten is het gebruikelijk om scope, einddatum en budget van een project van tevoren vast te leggen. Dit model wordt *fixed price* genoemd. In werkelijkheid is het vaker *fixed price*, *fixed scope* en *fixed date*. Fixed price lijkt nodig om duidelijke afspraken te kunnen maken tussen klant en opdrachtnemer. Ook veel aanbestedingen zijn hierop geschoeid. Dit lijkt plausibel. Maar in projecten waarin iedere fase precies één keer wordt doorlopen, moet iedere fase raak zijn. Compleet en volledig. En dat is moeilijk, zoals al eerder werd aangegeven.

SAMENWERKEN MET DE KLANT
OVER CONTRACTONDERHANDELINGEN

In agile wordt de scope niet van tevoren volledig gefixeerd. Dit biedt de klant de mogelijkheid om tijdens het project de requirements te wijzigen. Zo scheppen agile projecten ruimte voor voortschrijdend inzicht. Hoewel dit niet eenvoudig is, probeert agile zo vertrouwen tussen klant en opdrachtnemer te creëren in plaats van het contract dicht te timmeren.

Omgaan met veranderingen

Wanneer in een traditioneel fixed-price project iets onvoorziens gebeurt, levert dit altijd problemen op. Misschien leveren de analisten de requirements maar niet op. Of worden er tijdens het ontwikkelen nog nieuwe, maar cruciale requirements gevonden. Of komen tijdens de acceptatietesten grove fouten in requirements of software aan het licht. Voortschrijdend inzicht. Het is onvermijdelijk.

Des te later voortschrijdend inzicht voorkomt, des te groter de inspanningen zijn om het in te passen. Traditioneel worden wijzigingen dan ook niet direct geïmplementeerd, maar genoteerd in *change requests*. Zodra het project is afgerond, kan worden besloten deze wijzigingen te realiseren. Voortschrijdend inzicht wordt zo uitgebannen. Projecten realiseren dan wel de software die is afgesproken, maar vaak niet de software die nodig is.