

# Waardevolle software bestaat niet, op zichzelf

Jurgen Vinju

---

**Er is veel behoefte aan grip op software en de digitalisering die met software wordt bewerkstelligd. Eigenaren van softwaresystemen die veel impact (gaan) hebben – essentiële IT infrastructuur van bedrijven en overheden – moeten deze systemen continu aanpassen aan veranderende omstandigheden. Die aanpassingen vergen grip op de zaak: het kunnen hanteren van redelijke financiële en operationele risico's, tijdens het maken van grote stappen vooruit.**

Grip op software levert besparingen op. Er kunnen lagere kosten komen door automatisering en optimalisatie van processen. Grip op software levert ook groei op. Er kunnen hogere inkomsten komen door nieuwe of grotere markten. Dus betekent grip op software economische waarde voor de samenleving en meestal ook een betere kwaliteit van leven voor de mensen die ermee in aanraking komen. We hebben die grip nog steeds niet. Daar zijn duidelijk aanwijsbare redenen voor.

De mogelijkheden van software zijn eindeloos; ze zijn in theorie slechts beperkt door ons eigen bevattingsvermogen. Net zoals we met een computerspel de prachtigste fantasiewereld tot leven kunnen brengen, met haar eigen wetmatigheden, kunnen we met industriële software de grenzen van onze creativiteit en inventiviteit verkennen. Alleen die grens van ons bevattingsvermogen ligt veel dichterbij dan we ons realiseren. Dat we software kunnen maken, betekent niet dat we die software ook begrijpen en kunnen aanpassen aan nieuwe eisen en omstandigheden. Om grip te hebben op software is er veel meer begrip nodig over wat software nou eigenlijk is. Dat geldt in het algemeen, maar des te meer voor specifieke systemen.

## Legacy software: wat is het nog waard?

Er komen regelmatig urgente items in het nieuws over het plotselinge verlies van grip op software. Het gaat dan vaak over *legacy software*: hoge kosten en lage flexibiliteit. We zeggen dan bijvoorbeeld al snel dat COBOL een oude programmeertaal is die vervangen moet worden. De krantenkoppen spreken ook van ‘verouderde systemen’, zoals het railnetwerk van Engeland aan het eind van de industriële revolutie. De wet van de remmende voorsprong wordt aangehaald. Vernieuwing van verouderde technologie is meestal dé oplossing waar weinig alternatieven voor lijken te bestaan.

Deze manier van denken gaat voorbij aan de werkelijkheid van wat software nou eigenlijk is. De verhaallijn bevestigt keer op keer de vreemde gedachte dat softwaresystemen, op zichzelf, veel waarde hebben die onafhankelijk is van de expertise die erover bestaat. Het lijkt alsof softwaresystemen een soort auto’s zijn op een lopende band die nog verkocht moeten worden, of prachtig vastgoed dat alleen nog op een huurder staat te wachten. Dat is een utopie die een illusie van grip op de zaak projecteert.

Het *frame* dat software een opzichzelfstaand kwaliteitsproduct is dat kan worden opgeleverd, onderhouden en vernieuwd, mits je er maar geld voor hebt, komt vermoedelijk uit het grote ‘metaforenspeel’. Software is zo weinig tastbaar en inzichtelijk voor de meeste mensen, dat we vergelijkingen moeten maken met de andere gebieden waar we wél verstand van hebben. We noemen het *software engineering*, maar er is nergens een gebouw, motor of vliegtuig te bekennen. We praten over ‘software-architectuur’, maar er zijn niet of nauwelijks relevante blauwdrukken te bekennen, newtoniaanse wetten zijn irrelevant en er is nergens een vakjury voor prachtige softwarearchitecturen. De vergelijkingen slaan de plank mis, met als gevolg dat er misvattingen ontstaan die soms heel lang en heel wijdverspreid een eigen leven gaan leiden. Ook tijdens informaticaopleidingen worden ideeën de wereld in geholpen die studenten later tijdens hun werk denken te gaan waarmaken: en dat is het platonische idee van de maakbaarheid van ultieme kwalitatieve software. Als je maar slim genoeg bent, dan ga je een betere code schrijven dan al die noeste arbeiders die voor je kwamen. Soms kom je er pas jaren later achter dat dát juist een luchtkasteel is.

## **Van softwareproduct naar softwareproces**

Uitbaters van softwaresystemen moeten zich beter realiseren dat ze veel meer een langdurend proces managen dan dat ze een product op voorraad hebben. Dat is het proces van het constant onderhouden van het systeem door de experts die voorhanden zijn.

Als ze op Schiphol de continue investeringen in opleidingen en kennisdeling van lucht- en grondverkeersleiding zouden afschaffen omdat ze nou eenmaal zulke moderne (en dure) verkeerstorens hebben aangeschaft, dan zouden ze dezelfde vergissing begaan die wij al jaren met onze softwaresystemen begaan. Het proces komt dan op een moment tot een *grinding halt*, omdat we de motor ervan hebben stilgelegd. Die motor is menselijke expertise. Het gaat niet alleen om verstand van software in het algemeen, maar vooral om verstand van specifieke softwaresystemen, en de organisaties waar ze mee vervlochten zijn en de domeinkennis over de processen die aangedreven, gemeten en gecontroleerd worden door die systemen.

We vergeten namelijk drie essentiële en verbonden elementen van het ‘software zijn’ te onderkennen. Deze drie elementen zijn al wereldwijd bekend sinds eind jaren zeventig. Drie essentiële waarheden over software engineering die niet onderkend worden zijn:

- De waarde van software wordt met name bepaald door flexibiliteit om mee te veranderen. Software is een levende, zich alsmaar ontwikkelende, op een complexe manier samenhangende verzameling tekstuele en visuele documenten (Lehman, 1979). Softwarecomplexiteit is hierdoor omgekeerd evenredig met haar waarde. Ofwel, software die te moeilijk om te wijzigen is, voor de experts die voorhanden zijn, is niets meer waard.
- De kwaliteit van softwareveranderingen wordt bepaald door expertise over de gegeven software. Veranderen is een werkwoord. Als gevolg hiervan bestaat softwarewaarde (namelijk aanpasbaarheid) voor de helft uit de flexibiliteit van expliciete documenten (waaronder broncode) en voor de andere helft uit impliciete expertise (kennis, vaardigheden en attitudes) van de mensen die de software ontwerpen, schrijven, onderhouden, testen, en gebruiken (Lehman, 1979; Conway 1967).
- Softwarekwaliteit en softwareveranderingen vormen een vicieuze cirkel. In positieve zin, op basis van begrip, kan software meegroeien zonder al te veel com-

plexiteit toe te voegen, waardoor de waarde van het systeem steeds toeneemt. In negatieve zin leiden onbegrepen wijzigingen tot een neerwaartse spiraal van almaar toenemend onbegrip en complexiteit. Uiteindelijk wordt de bodem van een inflexibel onbegrepen en daarmee waardeloos systeem altijd een keer bereikt.

Ook al zijn dit een soort van ‘natuurwetten’ van de Software Engineering toch wordt de schuldvraag van het plotselinge en totale waardeverlies meestal belegd bij de technologie: de broncode, de programmeertaal, het operating systeem, het platform, de architectuur, de leveranciers van verouderde componenten, et cetera. Maar dat is (veel) minder dan de helft van het verhaal. Daarom halen we hier geen concrete voorbeelden aan: de verantwoordelijken die in het nieuws zijn geweest over *legacy software*, hebben meestal al genoeg te verduren gehad. Het inzicht is dit: waardevolle softwaresystemen bestaan niet zonder de menselijke expertise die noodzakelijk is om ze te onderhouden.

## **Softwareonderhoud is onvermijdelijk**

Dat onderhouden is inderdaad onvermijdelijk. Niemand heeft iets aan de Belastingdienst van vorig jaar. Sterker nog, die is juridisch onhoudbaar in het nieuwe jaar en moet worden aangepast. De software van de vorige machine van ASML is superwaardevol, mits die ook aangepast kan worden voor de nieuwe generatie machines. Anders is het einde oefening. Om van een *start-up* naar een *scale-up* te groeien, moet de onderliggende software mee veranderen zonder opnieuw te hoeven beginnen. De *venturecapitalist* die net het intellectuele eigendom heeft aangekocht, heeft er geen oren naar dat haar geld wordt uitgegeven om hetzelfde nog eens opnieuw te bouwen. Het heet software (zachte materie) omdat deze component, in tegenstelling tot hardware, verwacht wordt om precies te passen tussen de hardware en de mensen die het product of de dienst gaan gebruiken. De hardware gaat niet gemakkelijk veranderen, de mensen en hun eisen natuurlijk wel, en dus zal de software mee moeten veranderen.

Omdat alle software moet veranderen, zal alle software erg snel zijn waarde verliezen op het moment dat de relevante expertise niet (meer) aanwezig is om de software aan te passen. Met andere woorden: *legacy software* is alleen maar ‘oud’ en ‘inflexibel’ als er geen bijbehorende expertise meer bij aanwezig is. Als dit klinkt als een open deur, dan zijn we samen gelukkig uit het oude frame gestapt.

Hier zijn drie fictieve maar realistische voorbeelden (gebaseerd op echte anekdotes) over de invloed van expertise op de waarde van software.

- De briljante programmeur Linus Torvalds die de Linux kernel heeft geschreven, kan op basis van die kennis tientallen jarenlang samenwerken met anderen om die kernel te onderhouden op topniveau (Kernel x Torvalds = kassa). Als je dezelfde man vraagt om de Belastingdienst te onderhouden, dan zou hij daar niets van bakken. Oftewel: de Belastingdienst x Torvalds' kennis = nul komma nul.
- Een team van vijf COBOL-programmeurs is gepensioneerd en wordt teruggeroepen nadat een groot team van Java-engineers twee jaar lang heeft geprobeerd een legacy systeem te vervangen om het te kunnen uitbreiden met een nieuw feature (voor een heleboel geld). De vijf pensionado's concluderen na vijf dagen analyse dat het oude systeem de nieuwe *feature* eigenlijk al had, maar onder een andere naam. COBOL-systeem x COBOL-experts = veel geld.
- In 2015 won een programmeur in Delhi (India), van de honderdduizenden COBOL-onderhouders, de eerste prijs: Best Software Maintainer. Ze was het meest efficiënt en meest effectief in het oplossen van problemen en het toevoegen van features aan softwaresystemen uit Europa en Amerika die tientallen jaren oud zijn. Wat is haar geheim? Ze legt uit: 'Ik spendeer tachtig procent van mijn tijd aan het opsporen van de originele auteurs van de broncode. Wanneer ik ze eindelijk aan de lijn heb, dan vraag ik ze het hemd van het lijf. Daarna pas ik de code aan op basis van een accuraat begrip. Soms stellen ze zelf al de juiste wijzigingen voor.' De experts zelf aan het woord laten; dat was dus niet iedereen te binnen geschoten.
- In 2020 liet een grote Zwitserse verzekeraar al zijn COBOL-codes naar Java vertalen door een externe partij uit Amerika. Het systeem draaide daarna niet meer op een duur mainframe, maar op 'gewone' cloud-computers. De transformatie was honderd procent succesvol en alle systemen bleven werken. Bij de eerste nieuwe verzekeringsvorm die moest worden ingevoerd, bleek dat nieuwe databronnen moesten worden gekoppeld en een vers team van Java-programmeurs werd ingehuurd. Toen kwam men erachter dat het volume in broncode vervijfvoudigd was ten opzichte van de originele COBOL-code en dat de Java-code in een soort 'Jobol-' of 'Cava'-dialect was geschreven waar niemand meer wat van kon maken. Het systeem werd in één keer afgeschreven want niemand anders dan de partij uit Amerika wist nog hoe het ongeveer werkte. Zij rekenden een hogere prijs dan betaalbaar werd gedacht.

Kortom: de expertise rondom een softwaresysteem is eigenlijk altijd de sleutel tot succes, en het is daarmee ook de achilleshiel, het grootste risico, van de eigenaren. Zijn de software-engineers die het systeem goed kennen niet meer voorhanden, dan heb je dus wel echt een probleem. Nota bene: dat is een héél ander probleem dan ‘we hebben verouderde software’. Het heeft ook andere en vaak veel meer voordehandliggende oplossingen. Deze oplossingen hebben minder met de technologie van het systeem te maken. Ze hebben te maken met investeringen in de menselijke experts en hun omstandigheden: teams, taken en tools.

De extra investeringen die je als software-eigenaar kunt doen, of eigenlijk moet doen, om het expertise-niveau op peil te houden, zijn significant. Je kunt gerust rekenen op jaarlijks vijftien tot twintig procent van het totale budget dat ooit aan de software is uitgegeven. Vooral als de software jaarlijks blijft groeien in volume, blijven ook onderhoudskosten groeien. Maar dit moet vergeleken worden met de desinvesteringen van giga renovatie of vervangingsprojecten, die keer op keer maar in de prullenbak verdwijnen. Als je anders 20 of 40 miljoen euro zou weggooien, wat is dan 2 miljoen euro om verder te kunnen bouwen aan een succesvol team dat een succesvol softwaresysteem in de lucht houdt? Bovendien is elk jaar dat het bestaande systeem in de lucht blijft, aangepast en wel, ook kassa. Het alternatief is namelijk nogal kostbaar en risicovol. En als laatste, maar zeker niet als minst, kun je ook investeren in softwarekrimp: het opschonen en verwijderen van volume en complexiteit. Nou dát is een investering met rendement!

Kortom: grip op expertise levert grip op software op. Dit is een langetermijnvisie die zijn vruchten afwerpt. Organisaties worden daardoor ook minder afhankelijk van de waan van de dag, en kunnen gaan sparen voor een totale geplande renovatie zo eens in de twintig of dertig jaar, tenzij een systeem (én de bijbehorende expertise) in de tussentijd zo effectief is meeveranderd dat dat niet nodig is.

## **Streven naar perfectie 2.0**

Het nieuwe platonische ideaal is het softwaresysteem met een expertiseteam dat nooit wordt afgeschreven, maar desondanks onherkenbaar en op innovatieve wijze mee verandert in de tijd. Dat is iets heel anders dan het oude ideaal van het op zichzelf taande systeem dat voor nu en voor altijd perfect zal zijn. Beide zijn uiteraard onhaalbaar, maar één van de twee leidt tot onbegrip en de ander tot grip.

Maar wat is nou ‘software-expertise’ eigenlijk? Uit welke kennis, kunde, attitudes bestaat die expertise? En waar kun je die halen? Hoe houd je die kennis up-to-date? Over welke teams, taken en tools praten we over? We richten ons even op de verschillen met de standaard software human-resources-situatie:

- Software-experts wonen *in-house*; ze kennen niet alleen hun interne klanten en de kennisdomeinen op hun duimpje, maar voelen dezelfde urgentie en waarden van de organisatie waar de software voor is. Die waarden en kennis reflecteren in hun risicoanalyses en hun ontwerpbeslissingen. Ze staan niet alleen juridisch maar ook sociaal-economisch aan de kant van het systeem.
- Software-experts zijn analytisch sterk; ze weten dat ze niet alles (meer) weten, ook niet over het systeem dat ze zelf gemaakt hebben, maar ze kunnen bij elke vraag bedenken hoe ze het antwoord toch gaan vinden. Hun antwoorden zijn gewogen met een foutmarge, omdat ze weten wat ze überhaupt niet kunnen weten.
- Software-experts zijn ‘lui en slim’. Ze weten dat hun tijd beperkt is en dat ze grote effecten moeten behalen met weinig middelen. Daarbij kunnen ze bewezen moderne gereedschappen zoals versiebeheersystemen, *test driven development*, *digital twins*, code-generatie uit domeinspecifieke talen, *model driven engineering*, *reverse engineering (code-as-data)*, *continuous integration*, et cetera. effectief toepassen.
- Goede software-experts moeten wel een klein ego hebben. Ook al was hun werk uit het verleden geniaal, dan nog kan de tijd hen inhalen. Afscheid nemen van oplossingen uit het verleden en zelfs hele oplossingsstrategieën uit het verleden zijn een belangrijk kenmerk van de effectieve expert. Zulke experts ontvangen feedback over hun eigen expertise en de kwaliteit van de systemen die ze hebben ontworpen gracieus en doen er hun voordeel mee.
- Software-experts zijn zowel eeuwige studenten als eeuwige docenten. De nieuwsgierigheid naar wat er verandert in de context en wat er kan veranderen in het systeem is cruciaal. Ze kunnen leren van cocreatie met externe experts (adviesbureaus of leveranciers van speciale componenten) en elkaar helpen om te begrijpen waar het nou precies om gaat, zodat kennis en vaardigheden door de organisatie en door de tijd kunnen worden geborgd.

Deze specifieke lijst heeft vrij weinig met opleidingsniveau te maken of welke managementlaag. Het gaat om het eigenaarschap, de domeinkennis, de nieuwsgierigheid ten

aanzien van het systeem en de organisatie, de wil om gereedschappen toe te passen die werk uit handen kunnen nemen en de kracht om met feedback om te gaan. Dat kan én moet allemaal op alle niveaus.

Als we dit vanuit een team perspectief bekijken, dan zul je ten eerste expertise moeten verdelen over het systeem. Een bepaalde dekkingsgraad is noodzakelijk vooral in de uithoeken waar veel verandert. Elk versiebeheersysteem kun je bevragen over de huidige verdeling. Ten tweede moet er over de tijd expertise worden verlengd. Het klassieke *master-/apprentice*-systeem leent zich hier goed voor. Een dakpanconstructie die kwaliteit en continuïteit borgt.

Vanuit taakperspectief verandert er een heleboel tegelijkertijd. Als eerst je prioriteit was om brandjes te blussen met deadlines, dan kun je je als expert nu richten op het leren van de ins en outs van jouw hoek van het systeem. Komen er dan urgente taken, dan weet je snel aan wie je vragen moet stellen of welke documentatie en broncode er geanalyseerd moeten gaan worden. Je weet ook welke tools beschikbaar zijn om je urgente taak uit te kunnen voeren. Daardoor ben je sneller klaar met urgente taken en kun je je weer richten op de prioriteiten: het begrijpen en onderhouden van het systeem.

Met *tools* bedoelen we alle mogelijke gereedschappen die een software-expert kan gebruiken. Vaak is er door gebrek aan tijd geen vrijheid om zich nieuwe, of zelfs oude, gereedschappen eigen te maken. Dat is eigenlijk het omgekeerde van ‘expertise’. Dat de meeste broncode kan worden omgezet in bevroegbare databases (*code-as-data*), of zelfs gevisualiseerd, zodat er niet veel code gelezen meer hoeft te worden ‘met de hand’, of dat veel code gegenereerd kan worden met weinig inspanning uit leesbare bronnen (plaatjes of domeinspecifieke talen), zijn voor veel experts bevrijdende lessen die veel tijd vrijmaken voor de minder saaie, minder kwantitatieve, maar juist meer interessante en kwalitatieve kanten van het werk.

Over tools gesproken: ‘Ja, maar nu kunnen we software genereren en transformeren met LLMs. En dat scheelt dan toch enorm?’ In het oude frame maakt dit alles veel erger. Het is daar willekeurige code die niemand begrijpt, anders hadden ze het zelf wel ingetypt, en die niemand nu ook meer geschreven heeft. Zelfs in Delhi kunnen ze nu niemand meer bellen. Maar in het nieuwe frame kan dit zeer goed werken;

iemand die met verstand een prompt manipuleert en het resultaat kan analyseren op kwaliteit kan een hoop meters maken met een *chatbot*.

‘Ja, maar software onderhouden is niet onze kerntaak. Daarom kopen wij onze software en onze software engineers extern in. Dat is toch zo klaar als een klontje?’ Dat klinkt inderdaad zeer aannemelijk, maar alleen als we blijven geloven dat er überhaupt zoiets bestaat als waardevolle software zonder de bijbehorende expertise. Ook moeten we de lessen uit de jaren zeventig eventjes vergeten; namelijk dat softwaresystemen de organisaties waar ze voor zijn weerspiegelen en ermee vervlochten zijn. Stappen we uit die droom, dan is de unieke expertise over onze eigen softwaresystemen absoluut een *core value* en een *key enabler*.

- Software is een proces, geen product: de waarde van software ligt niet in de code zelf, maar in het vermogen om flexibel mee te veranderen met de organisatie.
- Menselijke expertise bepaalt softwarekwaliteit: zonder de juiste mensen om software te begrijpen en aan te passen, verliest zelfs de meest geavanceerde technologie zijn waarde.
- Grip op software is grip op expertise: duurzaam softwarebeheer vraagt niet om perfecte code, maar om een continu lerend en samenwerkend team dat systemen door de tijd heen onderhoudt en verbetert.