# De programmeur als kunstenaar

Bart Barnard

juni 2015

## Inleiding

Op het gebied van de design en kunst is een toenemende samenwerking met computerprogrammeurs waar te nemen. Bekende kunstenaar als Daan Roosegaarde of Rafaël Rozendaal of ontwerpers zoals Jonathan Puckey of Golan Levin maken intensief gebruik van de kennis en de vakmanschap van programmeurs om hun werk te laten bewegen of te reageren op activiteiten van bezoekers of toeschouwers. Het groeiende aantal bezoekers van Nederlandse festivals als STRP en DEAF is indicatief voor het fenomeen dat deze samenwerking goede vruchten afwerpt en zich op een toenemende populariteit mag verheugen.

Hoe goed deze samenwerking over het algemeen ook uitpakt en hoe bijzonder of mooi de resultaten hiervan ook zijn, het blijft toch nog vaak hangen in de gedachte van de kunstenaar als creatieveling en de programmeur als uitvoerder. In dit artikel breek ik een lans voor de gedachte dat het dagelijks werk van deze laatste, het daadwerkelijk *schrijven van programmacode* zich beter laat vergelijken met het werk van een (autonoom) kunstenaar dan met dat van een ambachtsman. Ik zal hierbij vier eigenschappen bespreken die vaak aan kunstenaar of kunstwerken worden toegeschreven — individualiteit, schoonheid, onzichtbaarheid, werkwijze — en aantonen dat deze eenvoudig kunnen worden getransponeerd naar het informatica-domein.

# Individualiteit

Het resultaat van het werk van een ambachtsman – een vaas, een schaal, een stoel – wordt in grote mate bepaald door de *functionele* werking hiervan en de *lichamelijkheid* van de personen die ermee moeten werken: de waarde van zo'n product hangt samen met onze *lichamelijke functionaliteit*(Boden, 2010, pp.50ff.). Hoe mooi of hoe leuk een product er ook uit ziet, als het niet doet waarvoor het gemaakt is, is het uiteindelijk geen goed product. De individualiteit die de ambachtsman in zijn werk kwijt kan, is derhalve gelimiteerd. Wil hij de functionaliteit (en dus de waarde) van zijn werk niet tenietdoen, dan moet hij zich hier beperken tot het toevoegen van ornamenten, kleine non-functionele versieringen.

Hoe anders is het werk van de autonoom kunstenaar. Zeker sinds de Renaissance hoeft hij zich niets gelegen laten liggen aan vorm of functie, maar kan hij in zijn werk al zijn creativiteit en individualiteit kwijt en het resultaat hiervan draagt – in ieder geval voor de kenners – ondubbelzinnig zijn handtekening: een schilderij van Giotto laat zich makkelijk onderscheiden van een schilderij van Mondriaan. Hetzelfde geldt voor de muziek en voor de literatuur: Mahler is anders dan Mozart, Marsman anders dan Multatuli.

Ditzelfde geldt voor de programmeur: in de programmacode kan hij een heel groot deel van zijn eigen individualiteit kwijt. Uiteraard moet de code doen wat het moet doen, maar de plasticiteit en de abstractie van het domein garanderen dat er voldoende ruimte bestaat om in de implementatie een volstrekt individuele stijl aan de dag te leggen. Het gaat hier niet, zoals bij de ambachtsman, om non-functionele ornamenten en additieven, maar om een fundamenteel verschil in stijl tussen de verschillende programmeurs. Net als kunstkenners kunnen veel programmeurs zonder moeite aan de stijl van de programmacode identificeren wie van zijn collega's dat specifieke stuk heeft geschreven. Programmeurs, zo lezen we ook bij Robert Martin, zijn *auteurs*:

> The `@author` field of a Javadoc tells us who we are. We are

> authors. And one thing about authors is that they have readers. Indeed, authors are responsible for communicating well with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort. (Martin, 2009, p.13f.)

Dat programmeurs auteurs zijn betekent ook dat zij zich een bepaalde eigen stijl kunnen aanmeten, één die door sommigen wel en door anderen niet gewaardeerd wordt. Deze stijl manifesteert zich op het niveau van implementatie, maar bijvoorbeeld ook op het architectuur-niveau. Welke *patterns* gebruikt deze programmeur? Hoe is het project opgezet? Allemaal keuzes die een individuele programmeur maakt en waarin hij zijn eigen persoonlijke voorkeuren en stijl kan laten gelden.

## Schoonheid

Over schoonheid is veel gezegd en geschreven. Schoonheid komt voor in de natuur, in mensen, en uiteraard ook in kunstuitingen. Wanneer ik iets schoon, iets mooi vind, zegt dat iets over mij: ik voel mij op een bepaalde manier verbonden met datgene wat ik ik mooi vind, zoek in mijn omgeving mensen op die hetzelfde mooi vinden en probeer mijn vrienden te overtuigen van de schoonheid van juist dit. Mensen die mijn esthetisch oordeel niet delen, hebben "het niet begrepen" of hebben een slechte smaak. Wanneer iemand iets wat ik heel mooi vind afkraakt, voelt dat als een aanval op mijn persoonlijkheid; hoewel *de gustibus non disputandum* appelleert dit toch aan een zekere objectieve status (Nehamas, 2010, p.85).

Eenzelfde situatie doet zich ook voor bij het schrijven van programmacode. Zo hebben programmeurs nagenoeg zonder uitzondering een duidelijke voorkeur voor een programmeertaal of een *framework* en proberen ze ook hun collega's te overtuigen van hun gelijk. Argumenten om de ene boven de andere taal te verkiezen beroepen zich vaak op de *schoonheid* van deze taal, zijn transparantie en leesbaarheid. Programmeurs zoeken collega's op

om met elkaar over ontwikkelingen van hun favorieten taal te spreken en er zijn online voldoende *flame wars* te vinden over de beste taal en het beste *framework*.

Ook op architectuurniveau wordt gesproken in termen van schoonheid – soms zelfs in die mate dat functionaliteit of werking moet wijken voor een *mooiere* implementatie – uiteraard geldt hier opnieuw de individuele smaak van de programmeur. Zo vinden de meeste collega's de recursieve implementatie van de Ackermann-functie die gegeven is in Listing 1 *mooier* dan die in Listing 2, hoewel de eerste loopt in $\Omega(\sqrt{2})$ en de tweede in $\Theta(|V| + |E|)$. Net als kunstwerken kunnen implementaties *mooi* zijn en net als kunstenaars worden goede programmeurs geroemd om de *schoonheid* van hun werk.

**Listing 1** Ackermann-functie (recursief)

```
function msRect(A,i)
    if i ≥ len(A) then
        return 0
    else
        return max(msRect(A, i+1), msRect(A, i+2))
    end if
end function
```

**Listing 2** Ackermann-functie (lineair)

```
function msRectIter(A)
    n ← len(A)
    soIns ← [o] * (n + 1)
    soIns[1] ← A[n − 1]
    for k ← 2, n + 1 do
        soIns[k] ← max(soIns[k − 1], A[n − k] + soIns[k − 2])
    end for
    return soIns[n]
end function
```

## Onzichtbaarheid

Een mogelijke tegenwerping tegen het idee van de programmeur als kunstenaar zou kunnen zijn dat de materie van het kunstwerk altijd zichtbaar blijft, terwijl de programmacode niet in het uiteindelijke product te zien is. Zo gauw de programmeur zijn werk heeft afgerond, als de app of het site helemaal klaar en operationeel is, is noodzaak of de behoefte om de code die het geheel laat werken zichtbaar te maken verdwenen. Dit in tegenstelling tot het traditionele kunstwerk: het blijft altijd mogelijk een schilderij te bekijken op het niveau van de verf, een muziekstuk kan ontrafeld worden op het niveau van de individuele noten en een gedicht bestaat in laatste instantie uit letters en woorden. Hoewel de *boodschap*, de *betekenis* van het werk zich op een ander niveau bevindt, is een dergelijke reductie tot het puur fysieke domein mogelijk en in veel gevallen zelfs gewenst. Veel bezoekers bekijken kunstwerken op deze verschillende niveau's, terwijl geen enkele app-gebruiker zelfs maar de mogelijkheid heeft om de programmacode die de app mogelijk maakt in te zien (behalve de ontwikkelaar zelf uiteraard).

Veel kunstwerken bestaan echter juist dankzij hun onzichtbaarheid. Neem het werk *Vertical Earth Kilometer* van Walter de Maria: een cilindrisch gat van vijf centimeter doorsnede en een kilometer diep dat geboord is in Kassel en vervolgens volgestopt met duizend koperen buizen van een meter lang. Het enige wat van dit werk zichtbaar is, is het deksel op het Friedrichsplatz Park dat het gat afsluit: de buizen zelf worden nooit meer gezien en het werk bestaat uit de *wetenschap* van wat er zich hier onder bevindt. Op een vergelijkbare manier kan een app gezien worden als een deksel, een zichtbare afsluiting van een complex maar onzichtbaar achterliggend systeem. Andere kunstwerken bestaan zelfs uitsluitend door iets wat er niet is: denk hier bijvoorbeeld aan het bekende *4'33"* van John Cage, een muziekstuk zonder geluid van exact die lengte. Of *Double Negative* van Michael Heizer, een werk dat bestaat bij gratie van de *afwezigheid* van rotsen in twee tegenovergelegen stukken berg in Nevada.

Iets hoeft, kortom, niet per se waarneembaar te zijn om als kunstwerk

gekwalificeerd te kunnen worden. Net als bij programmacode is het uiteindelijke werk vaak maar een heel klein zichtbaar deel van een veel groter onderliggend geheel.

## Werkwijze

Een fundamentele eigenschap van het creatieve proces van elke werkelijke kunstenaar is de fluïde werkwijze. Voorafgaand aan het werk heeft de kunstenaar een vaag omlijnd idee van wat er ongeveer moet gebeuren, maar pas in de *dialoog* met het materiaal wordt duidelijk wat er precies gaat gebeuren. Bekend is de uitspraak van Michelangelo dat het beeld al in het marmer aanwezig is en dat je alleen maar het overbodige marmer hoeft weg te slaan; een vergelijkbare uitspraak is dat het werk de kunstenaar gebruikt om zichzelf te kunnen manifesteren. Bij de ware kunstenaars verdwijnt het onderscheid tussen werk en persoon maar vormen deze twee samen juist een eenheid. De kunstenaar wordt als het ware het medium waardoor het kunstwerk tevoorschijn komt (Heidegger, 2003, p.26).

Iets vergelijkbaars geldt voor de programmeur. Hoe goed de architectuurplaatjes ook zijn, hoe fijnmazig de planning ook is opgezet en hoe nauwkeurig ook de *use cases* ook zijn omschreven, pas wanneer er daadwerkelijk programmacode geschreven wordt ontstaan er dingen en wordt duidelijk wat er precies gaat gebeuren. Net als de kunstenaar komt de programmeur onvoorziene en onvoorzienbare zaken op zijn pad tegen die hem dwingen het werk anders in te richten of anders op te zetten. Net als de kunstenaar gaat de programmeur een dialoog aan met zijn werk; de weerbarstigheid van de materie (verf of marmer in het geval van de kunstenaar, hard- en software in het geval van de programmeur) heeft tot gevolg dat het nooit op voorhand volledig duidelijk kan zijn wat er exact gaat gebeuren wanneer de eerste regels code worden geschreven of nieuwe functionaliteit aan een legacy-systeem wordt toegevoegd. De programmeur gaat op in zijn werk en vormt hiermee een creatieve eenheid (Dreyfus & Kelly, 2011, p.81).

# Conclusie

Uiteraard gaat het bij het schrijven van programmacode om functionaliteit en werkzaamheid – een programma of een app moet natuurlijk doen wat het moet doen. Echter de formulering van datgene wat het moet doen, en de wijze waarop dat geïmplementeerd wordt, worden pas duidelijk op het moment dat de programmeur daadwerkelijk met het programmeren aan de slag gaat. Wanneer we, met Kent Beck, stellen dat de *verandering* de enige constante is in het schrijven van computerprogramma's (Beck, 2004), moeten we deze verandering incorporeren in de dagelijkse praktijk van de programmeur. En dat betekent dat we moeten erkennen dat de programmacode zelf net zo goed de handelingen van de programmeur bepalen als de programmeur de vorm van de programmacode; dat deze twee, kortom, met elkaar in dialoog treden om een gezamenlijk doel te bereiken. Dat het maken van een computerprogramma zich beter laat vergelijken met het houwen van een beeldhouwwerk dan met het maken van een brug; dat programmeren een *creatieve aangelegenheid* is.

Als docenten en begeleiders moeten we onze studenten derhalve voorbereiden op de verandering, op het feit dat software-ontwikkeltrajecten zich net zo min volledig laten plannen als het maken van een kunstwerk. We moeten kijken naar de creatieve opleidingen en industrieën, onderzoeken welke aspecten daarvan we in onze curricula kunnen overnemen, en inventariseren op welke punten hiermee samenwerking mogelijk is. Want de computer-expert die zichzelf louter profileert als technologisch ingenieur, en het hele creatieve moment dat inherent in dit werk aanwezig is uit het oog verliest, ontneemt zichzelf en zijn omgeving de mogelijkheid werkelijk grootste daden te verrichten.

# Literatuur

Beck, K. (2004). *Extreme programming explained: Embrace change.* Addison-Wesley Longman Publishing Co., Inc.

Boden, M. A. (2010). *Creativity & art. three roads to surprise.* Oxford (UK): Oxford University Press.

Dreyfus, H. & Kelly, S. D. (2011). *All things shining. reading the western classics to find meaning in a secular age.* New York: The Free Press.

Heidegger, M. (2003). Der ursprung des kunstwerkes. In M. Heidegger (red.), *Holzwege.* Frankfurt aM: Vittorio Klostermann, GmbH.

Martin, R. C. (2009). *Clean code. a handbook of agile software craftsmanship.* Boston, MA: Pearson Eduction, Inc.

Nehamas, A. (2010). *Only a promise of happiness. the place of beauty in a world of art.* Princeton, NJ: Princeton University Press.