

De Realiteit van Denkniveaus bij Algoritmen

Jacob Perrenet, Jan Friso Groote en Eric Kaasenbrood

SAMENVATTING

Eerder werd in TINFON verslag gedaan van onderzoek aan de Technische Universiteit Eindhoven naar abstractieniveaus in het denken studenten over algoritmen (jaargang 13, nr. 4). Vier niveaus werden verondersteld, waarvan de middelste twee duidelijk werden gevonden bij een groep van 200 bachelorstudenten Informatica en waarbij daarnaast ook groei zichtbaar was van het ene naar het andere niveau. In dit artikel wordt vervolgonderzoek beschreven met diepte interviews bij een kleiner aantal studenten. De resultaten geven kwalitatieve steun aan de realiteit van de niveaus. We toetsen de indeling ook aan andere indelingen¹ en leggen de link met het onderwijs.

Didactisch onderzoek nodig naar begripsontwikkeling in de Informatica

In een eerder nummer van dit blad pleit Van Diepen (2005) voor onderzoek naar de moeilijkheden bij abstractie en generalisatie in het programmeeronderwijs (in het voortgezet onderwijs). Het zou goed passen in het geplande onderzoeksprogramma voor de te starten lerarenopleiding. Wij zien een nauwe verwantschap van deze plannen met ons onderzoek naar abstractieniveaus in het begrip van algoritmen in het hoger (universitair) onderwijs, dat aansluit bij internationaal didactisch onderzoek. De bestaande traditie van onderzoek naar begripsniveaus in het wiskundeonderwijs kan daarbij inspireren (Almstrum, e.a., 2002).

Niveautheorie

Goed toepasbaar bij begrippen uit de Informatica blijken niveautheorieën zoals van Van Hiele (Van Hiele, 1986) en Skemp (Tall en Thomas, 2002), getuige het werk van bijvoorbeeld Hazzan (2002), Aharoni (2000) en Haberman e.a. (2005). De mate van abstractie van een denkniveau heeft meerdere aspecten, maar het meest fundamentele is abstractieniveau als uitdrukking van de verhouding tussen proces en object. Iemand bereikt een hoger abstractieniveau ten aanzien van een concept, wanneer processen en relaties tussen objecten, die verband houden met dat concept, op zich als een objecten beschouwd kunnen worden beschouwd. In ons eerdere onderzoek werkten we dit uit voor het begrip algoritme (Perrenet e.a., 2004, 2005a en 2005b).

Niveaus bij het begrip algoritme

We construeerden de volgende vier niveaus:

1. Het executieniveau: het algoritme is een specifieke run op een concrete specifieke machine; de benodigde tijd voor uitvoering wordt door die machine bepaald.
2. Het programmaniveau: het algoritme is een proces, beschreven door een specifieke, uitvoerbare programmeertaal; de uitvoeringstijd hangt af van de input.
3. Het object niveau: het algoritme staat los van een specifieke programmeertaal; het wordt niet meer als proces, maar als object gezien; bij de constructie van een algoritme worden datastructuur en invariantieeigenschappen gebruikt; metaeigenschappen, zoals terminatie, zijn relevant en de benodigde tijd wordt beschouwd in termen van orde grootte als functie van de input.

4. Het probleemniveau: het algoritme kan als een object worden gezien met de eigenschappen van een 'black box'; het perspectief is geworden: gegeven een probleem, welk type algoritme is geschikt? Problemen kunnen worden gecategoriseerd volgens geschikte algoritmen; een probleem heeft een intrinsieke complexiteit.

We veronderstelden dat studenten in de loop van hun studie groeien in hun niveau. Bij het bereiken van een hoger niveau gaat het oude niveau niet verloren, maar wordt opgenomen in het nieuwe geheel. Een probleem kan uit de beschikbare niveaus een bepaald niveau van denken oproepen.

Vragenlijst

We construeerden een vragenlijst om herhaaldelijk bij verschillende groepen bachelors het niveau te meten, bestaande uit zeven items. De lijst begint met het volgende item:

0. Geef je definitie van algoritme.

Dit wordt gevolgd door zes items in de vorm van een vraag of men het eens is met een bepaalde uitspraak (de gegeven alternatieven zijn: eens, oneens, eens en oneens kan beide, weet niet) plus het verzoek om *argumenten* voor het gekozen alternatief.

1. Een algoritme is een programma, geschreven in een programmeertaal.
2. Twee verschillende programma's in dezelfde programmeertaal kunnen implementaties zijn van hetzelfde algoritme.
3. De correctheid van een algoritme is in het algemeen te bewijzen door de implementatie te testen met slim gekozen testcases.
4. Een geschikte maat om te meten hoe lang een bepaald algoritme erover doet om een bepaald probleem op te lossen is de tijd die het kost in milliseconden.
5. De complexiteit van een probleem is onafhankelijk van de keuze van het algoritme waarmee je het oplost.
6. Bij ieder probleem is het mogelijk dat in de toekomst algoritmen worden gevonden die een ordegrrootte efficiënter zijn dan de nu bekende.

Op basis van de vier veronderstelde niveaus werd een gedetailleerde scoringslijst ontwikkeld, gericht op analyse van de gegeven argumenten. Dus niet de keuze maar de aard van de argumenten bepaalde het niveau.

Aanwezigheid en groei van niveaus

Als belangrijkste resultaten vonden we:

- niveau 2 en 3 zijn duidelijk aanwezig
- in een hoger jaar is het niveau bij de meeste studenten ook hoger
- binnen een jaar groeit het niveau bij de meeste studenten

De resultaten bleken betrouwbaar, maar de validiteit was nog niet duidelijk: in hoeverre was werkelijk begripniveau gemeten? Was het niet mogelijk dat de studenten slechts de juiste termen gebruikten, maar bij nader doorvragen door de mand zouden vallen. Als aanvulling op het eerste vooral kwantitatieve onderzoek werd er daarom kwalitatief onderzoek naar gedaan of studenten de door hen gebruikte termen werkelijk begrepen. De combinatie van kwantitatief onderzoek met veel studenten plus kwalitatief onderzoek met weinig studenten is aan te bevelen (zie bijvoorbeeld Almstrum, 2005).

Interviews

Negen Bachelor studenten uit verschillende jaargangen werden gevraagd de vragenlijst hardop denkend in te vullen en vervolgens geïnterviewd op basis van de gegeven antwoorden. Inclusief een korte training in hardop denken nam een sessie ongeveer 50 minuten tijd. Het hardopdenken maakte enerzijds 'de tong los' voor het latere interview en gaf anderzijds de interviewer meer directe informatie voor het stellen van vragen. Het eerdere onderzoek had ons namelijk geleerd welk type antwoorden konden worden verwacht. Daardoor konden interviewvragen vooraf naar maat worden opgesteld. Als voorbeeld geven we enkele typische antwoorden voor item 3:

De correctheid van een algoritme is in het algemeen te bewijzen door de implementatie te testen met slim gekozen testcases (Eens, oneens, eens en oneens kan allebei, ik weet het niet).

Antwoord 1: Eens; uitputtend testen geeft je een goed idee over de correctheid.

Vragen: *Hoe doe je dat, uitputtend testen? Welke testen doe je dan? Wat bedoel je met een 'correct' programma? Hoe weet je, dat bij implementatie geen eigenschappen worden toegevoegd die de testresultaten beïnvloeden? Hoe weet je dat je alle mogelijkheden hebt getest?*

Antwoord 2a: Oneens; uitputtend testen is vaak moeilijk; je moet het bewijzen.

Antwoord 2b: Oneens; je moet het bewijzen met de pre-/post-conditie techniek.

Vragen: *Waarom is uitputtend testen vaak moeilijk? Wat moet er volgens jou precies bewezen worden? Waarom zou bewijzen tot een beter resultaat leiden; wat bereik je met bewijzen? Heb je een garantie dat de implementatie van een correct algoritme correct functioneert?*

Het was de taak van de interviewer geen nieuwe inhoudelijke informatie te geven.

Analyse en resultaten

De interviews werden opgenomen en uitgetypt. Uit de protocollen werd voor alle door de studenten gebruikte termen vastgesteld of de term al of niet werd begrepen. Deze analyse gebeurde per item, geanonimiseerd en gerandomiseerd. In Tabel 1 zijn de resultaten gegeven. De symbolen +, ± en – hebben de volgende betekenis: + de student begrijpt de term goed of redelijk goed; - de student begrijpt de term niet; ± het is niet helemaal duidelijk of de student de term begrijpt. Een lege cel geeft aan dat de student de term niet gebruikt in de vragenlijst of in het interview. De letters e, t en d staan voor eerstejaars, tweedejaars en derdejaars. Het belangrijkste resultaat is dat de studenten meestal de termen begrijpen die ze hanteren. Bij slechts 5% van het gebruik is onvoldoende begrip geconstateerd, bij 20 % bleef er enige twijfel over, maar bij 75% van het gebruik is het begrip (redelijk) goed. Daarmee zijn de resultaten van het eerste onderzoek ondersteund.

We willen er wel op wijzen dat we deze resultaten niet buiten de verzameling van informaticastudenten durven te generaliseren. We denken hierbij speciaal aan studenten van andere opleidingen die service onderwijs informatica volgen. We geven een voorbeeld. Een docent vertelde ons, dat hij bedrijfskundestudenten bij lineaire programmering het simplexalgoritme onderwees. Ze leerden dat deze oplossingstechniek niet polynomiaal is en dat er ook een polynomiaal algoritme bestaat dat echter in de praktijk slechter werkt. Het lijkt nu, alsof deze studenten direct op het hoogste niveau denken. We vermoeden – maar dat zou onderzocht moeten worden – dat deze studenten eerder dan de informaticastudenten een onterecht hoge niveauscore op de vragenlijst zouden krijgen en dat ze bij een interview wel 'door de mand zouden vallen'. We vermoeden dat deze studenten niet in staat zullen zijn onderliggende niveaus op te roepen; ze zullen alleen in standaardsituaties weten hoe te

handelen, maar niet in nieuwe situaties. Ze zullen het inzicht missen, dat wel aanwezig is bij studenten waar de niveaus een voor een zijn opgebouwd. Het onderscheid tussen deze twee soorten van denken kunnen we benoemen als *instrumenteel* begrijpen (weten *hoe*) tegenover *relationeel* begrijpen (weten *waarom*). Dit is een klassiek onderscheid afkomstig uit de wiskundendidactiek (zie Skemp, 1976).

Tabel 1. Gebruikte termen en mate van begrip van die termen

Term	e1	e2	e3	T1	t2	d1	d2	d3	d4
algoritme	+	+	+	+	+	+	+	+	+
implementatie	+	+	+	+	+	+	+	+	+
programmertaal	+	+	+	+	+	+	+	+	+
programma's die verschillen	+	+	+	+	+	+	+	+	+
correctheid	+	±	+	+	+	+	+	±	+
bewijzen	+	±	+	+	+	+	+	±	+
bewijstechniek	+	+	+	+	±	±	+	+	+
complexiteit van een probleem	±	±	±	-	±	+	±	-	+
complexiteit van een algoritme	±	+	±	±	+	+	+	+	+
ordegrootte	+	±	±	±	+	+	+	+	+
GCL ²			±	±		±			+
pre/post conditie			+					+	+
NP				+		-		±	
specificatie			±		+				
abstract						+			+
quantum computer				-					-
syntax							-	+	
ontwerp					+				
stappen		+							
definitie		+							
concept									+
formele taal				+					
precies						+			
compiler				±					
natuurlijke taal								+	
semantiek									+
-schema ³				-					
state						+			
assembler instructie				+					
upper bound		±							
greedy					±				
dynamisch programmeren					±				
lower bound								+	
P								+	

Wanneer we tabel nader bekijken, zien we dat het vooral bij de term complexiteit het lastig bleek binnen de gestelde tijd tot een oordeel over het al of niet begrijpen te komen. Verder valt op dat terwijl sommige termen door alle studenten gebruikt worden (de belangrijkste termen uit de tekst van de vragenlijst), er ook grote individuele variatie is wat betreft de

termen die worden genoemd. Dat laatste wijst op grote individuele variatie in de onderliggende denkstructuren. Uit de interviews kwam naar voren, dat zelfs studenten uit dezelfde jaargang zich soms zelfs heel verschillend uiten wanneer het gaat om hetzelfde concept. Juist kwalitatief onderzoek is in staat dergelijke variatie zichtbaar te maken. Zie voor voorbeelden Perrenet en Kaasenbrood (in print).

Universitaire docenten over abstractie bij algoritmen

Het is interessant te om verschillende classificaties van abstractie bij het begrip algoritme te vergelijken met de onze. Bij ons zijn twee dergelijke classificaties bekend. In de volgende sectie bekijken we Van Diepens classificatie van de moeilijkheden bij het programmeren. In deze sectie bekijken we de resultaten van een ambitieus project aan de TU/e om van alle opleidingen het academisch profiel te meten (verder Meijers e.a. 2005 en de website van het Platform Academische Vorming). Aan enkele informaticadocenten werd – los van het hier beschreven onderzoek – gevraagd op vier dimensies (abstract, concreet, synthetisch en analytisch) niveaus in het onderwijs te definiëren aan de hand van series in complexiteit toenemende casussen. Voor de dimensie ‘abstract’ werd de volgende definitie gehanteerd: *Abstraheren is het op een hoger aggregatieniveau brengen van een beschouwingswijze (uitspraak, model, theorie), waardoor deze op meer gevallen van toepassing kan zijn. Een beschouwingswijze is abstracter naarmate het aggregatieniveau hoger is.* Hiermee kwamen ze tot maar liefst zes niveaus en wel de volgende:

1. Een programma, gericht op het oplossen van een bepaald probleem, dat draait op een gegeven machine op een gegeven moment. Tijdmetering als de doorlooptijd van het programma.
2. Een programma dat draait op een gegeven machine onder geïdealiseerde omstandigheden. Tijdmetering als zuivere executietijd in klokseconden.
3. Een programma opgevat als serie elementaire berekeningen in een programmeertaal, draaiend op een geïdealiseerde machine. Tijdmetering als bestede aantal elementaire berekeningen in termen van de programmeertaal.
4. Een programma opgevat als serie elementaire berekeningen in een wiskundige taal, draaiend op een Turing-machine. Tijdmetering als het bestede aantal elementaire berekeningsstappen.
5. Een programma opgevat als serie elementaire berekeningen in een wiskundige taal (draaiend op een Turing-machine) met alle mogelijke invoer. Tijdmetering als orde-grootte van het aantal benodigde berekeningstappen als functie van de orde-grootte van de invoer (i.e. de orde-grootte van complexiteit van het algoritme).
6. Alle mogelijke algoritmen die het probleem oplossen. Tijdmetering als orde-grootte van de complexiteit van het probleem.

Typisch is dat hier abstractie voornamelijk gezocht wordt in het kiezen van een abstracter machinemodel. Wanneer we de Turing-machine nog als een computer opvatten, overstijgen de niveaus 1-4 nauwelijks ons executieniveau. Het niveau 5 lijkt vooral overeen te komen met ons programmaniveau: pas daar worden verschillende soorten input beschouwd. Niveau 6 past precies op het probleemniveau. Het objectniveau, waarbij een algoritme als een te manipuleren en te bestuderen object gezien wordt en de executeerbaarheid nauwelijks van belang is, is moeilijk in de indeling terug te vinden. Misschien in het loslaten van de programmeertaal ten gunste van de wiskundige taal.

We trekken twee conclusies. Enerzijds bestaat er nog geen gestandaardiseerde hiërarchie van niveaus over algoritmische kennis. Verschillende deskundigen komen tot verschillende indelingen. Anderzijds is de bovenstaande indeling, ondanks de verschillen, zodanig conform onze indeling, dat we hierin een argument voor de natuurlijkheid van onze indeling zien. De overeenkomst ondersteunt de validiteit van onze meetmethode.

Vanuit didactisch oogpunt is het feit dat de meeste docenten zich kunnen vinden in het idee van abstractieniveaus hoopgevend. Ze zullen zich realiseren, dat de termen die ze gebruiken afhankelijk van het niveau voor hun studenten een andere betekenis kunnen hebben en dat het tijd en moeite kost het verschil te overbruggen. Het risico dat studenten wel de termen leren maar niet het bijbehorende inzicht zal in het hoger onderwijs vooral groot zijn in het serviceonderwijs, zoals we in de vorige paragraaf illustreerden.

Indeling moeilijkheden bij het leren programmeren

We gaan terug naar het artikel van Van Diepen (Van Diepen, 2005), waaraan we in het begin refereerden. Vanuit eigen onderwijservaring geeft hij een indeling van begripsmatige problemen bij het leren programmeren, die te maken hebben met abstractie en generalisatie. Deze begripsmatige problemen moeten overwonnen worden om van een niveau naar het andere te gaan. Aan de hand van een casus ‘sorteren van een ledenbestand’ komt hij tot:

1. Datarepresentatie op recordniveau (of classniveau)
2. Data-abstractie
3. Datarepresentatie op collectieniveau
4. Procedurele abstractie: ontwerpniveau (sorteren als één actie)
5. Procedurele abstractie: hergebruik
6. Procedurele abstractie: black box
7. Procedurele abstractie: parameter mechanisme
8. Procedurele generalisatie van de probleemomvang
9. Procedurele generalisatie van datatype
10. Procedurele generalisatie van het probleem
11. Generalisatie van het algoritme.

We nemen 1, 2, 3 en 9 die over datatypes gaan niet in beschouwing. Bij de categorieën rond het begrip data zullen we het eerder moeten zoeken bij een abstractieniveau-indeling van *datastructuren*. Aharoni (2000) constateerde overigens in interviews met studenten drie in abstractie oplopende niveaus van denken over datastructuren: programmeertaal-georiënteerd denken, programmeren-georiënteerd denken en programmeer-onafhankelijk denken. Deze driedeling vertoont overigens wel duidelijke verwantschap met de onze.

De andere overgangen hebben betrekking op het niveau van algoritmisch denken. Onderdeel 4 lijkt vooral bedoeld om studenten naar het objectniveau te tillen. De onderdelen 5, 6, 7, 8 en 10 zijn bedoeld om bewustzijn aan te leren over het feit dat programma's voor meerdere invoer geschikt moet zijn. Daarmee worden de studenten op programmaniveau gebracht. 11 is duidelijk bedoeld om studenten te laten abstraheren van een bepaald algoritme, en te denken in termen van problemen: het probleemniveau. Er valt op dat er vooral veel aandacht is voor de lagere niveaus van de hiërarchie. Zelfs 11 wordt door Van Diepen uitgelegd in termen van programmeertaalconstructies. We kunnen een groot deel van zijn categorieën in onze ordening inpassen. Ook dit geeft redelijke steun aan de validiteit van onze indeling.

Onderzoek nodig

Van Diepen besluit zijn artikel met de wens dat zijn begripsmatige probleemgebieden opgenomen worden in het didactiekonderzoeksplan van de nieuwe lerarenopleiding Informatica. We zijn het met hem eens. Om te beginnen vraagt zijn inventarisatie dan om

nadere structurering. We hebben een eerste stap gedaan met de ordening naar onze abstractieniveaus, maar konden daarbij niet alle probleemgebieden plaatsen. Ook zijn we voorbijgegaan aan – hier niet genoemde – aspecten van de problemen die meer te maken hebben met analyseren van de probleemsituatie (*analyse*), het ontwerpen van de oplossing (*synthese*) en de concrete uitwerking (bijvoorbeeld *concretisering* in een specifieke programmeertaal). We moeten ons realiseren dat de moeilijkheid van problemen niet alleen met *abstractie* te vangen is. Abstractie is slechts één dimensie van de moeilijkheid van problemen (of van het denken daarover). Met de vier dimensies (abstract, concreet, analyse en synthese) van het eerder genoemde Eindhovense Project Academische Vorming hebben we een uitgebreider instrumentarium in handen. We kunnen de mogelijkheden voor didactische toepassing daarvan hier vanwege de beschikbare ruimte niet verder onderzoeken, maar achten het de moeite waard er t.z.t. in een Tinfon-artikel of een NIOC-presentatie op terug te komen.

Het categoriseren van problemen van leerlingen en het ontdekken van niveaus is natuurlijk slechts een eerste stap naar theorieontwikkeling over begripsvorming in de informatica en didactische toepassingen daarvan. In Nederland zijn we daar nog maar net mee begonnen; de start van de lerarenopleiding zal het verder op gang brengen. Gelukkig hoeven we niet alles van de grond af op te bouwen. Verwezen werd al naar informaticadidactisch onderzoek buiten Nederland; de Association for Computing Machinery (ACM) heeft een Special Interest Group voor Computer Science Education met de tweejaarlijkse ITiCSE-conferentie (zie de website in de literatuurlijst). Genoemd werd ook de relevantie van het onderzoek binnen de wiskundendidactiek. Maar wat te denken van de al vele jaren bestaande Psychology of Programming Interest Group met jaarlijkse werkgroepbijeenkomsten (zie ook de site in de literatuurlijst)? En ook de onderwijspsychologie in het algemeen kan vruchtbare perspectieven leveren. Het overzicht van Van Diepen geeft de indruk, dat de moeilijkheid ook vooral kan zitten in de *combinatie* van problemen. Technische problemen, taalproblemen, ontwerpproblemen, precisieproblemen bij syntax en semantiek, perspectiefwisselingen, niveausprongen: er speelt te veel tegelijk! De zogeheten Cognitive Load Theory (zie bijvoorbeeld Merriënboer e.a. 2005) komt tot voorstellen voor opdrachtvormen om de geheugenbelasting acceptabel te maken, zodat de leerling ondanks de bomen het bos weer kan zien.

Slot

We hebben in nader onderzoek kwalitatieve ondersteuning gevonden voor onze niveau indeling van abstractie in het denken van studenten over algoritmen. Andere indelingen zijn mogelijk, maar dezelfde dimensie blijft herkenbaar. Niveau-indelingen en een theorie over niveauverhoging kunnen een bijdrage leveren aan het oplossen en voorkomen van problemen van leerlingen en studenten met programmeren en het creëren en bestuderen van algoritmen, maar dit perspectief is zeker niet voldoende. Meerdere dimensies dan abstractie alleen zijn relevant. Nederlands didactisch onderzoek gericht op de informatica kan profiteren van het werk dat al gedaan is buiten Nederland en ook buiten de informaticadidactiek: bij de wiskundendidactiek en bij de onderwijspsychologie.

Noten

¹ We danken Tijn Borghuis voor het kritisch meedenken bij dit laatste deel van het artikel.

² GCL = Guarded Command Language, a formele taal voor het systematisch construeren van correcte programma's, een overblijfsel uit de zogeheten Dijkstra-periode van de Informatica aan de TU/e; zie ook Kaldewaij (1990).

³ _-schema = Een algoritmische ontwerpmethode in GCL met gebruik van invarianten.

Literatuur

- Aharoni, D. (2000). Cogito, Ergo, Sum! Cognitive Processes of Students Dealing with Data Structures. *Proceedings SIGCSE*, Austin, , 26-30.
- Diepen, N. van (2005) Elf redenen waarom programmeren zo moeilijk is; *Tijdschrift voor informaticaonderwijs*, 14e jaargang, nummer 4, 105-107.
- Almstrum, V.L., e.a. (2002). Import and Export to/from Computing Science Education: The Case of Mathematics Education Research. *Proceedings ITiCSE*, Aarhus, 193-194.
- Almstrum, V.L., Guzdial, M., Hazzan, O. and Peter M. (2005). Challenges to Computer Science Education Research. *Proceedings ITiCSE*, St. Louis.
- Haberman, B., Averbuch, H. and Ginat, D. (2005). Is it really an Algorithm? – The Need for Explicit Discourse. *Proceedings ITiCSE*, Monte de Caparica, 74-78.
- Hazzan, O. (2002). Reducing Abstraction Level when Learning Computability Concepts. *Proceedings ITiCSE*, Aarhus, 156-160.
- Kaldewaij, A. (1990). *Programming: The Derivation of Algorithms*. Prentice Hall International, UK.
- Meijers, A.W.M., C.W.A.M. Overveld & J. Perrenet (2005) *Criteria voor academische Bachelor and Master curricula*, Technische Universiteit Eindhoven.
- Merrienboer, J.J.G. en J. Sweller (2005). Cognitive Load Theory and Complex Learning: Recent Developments and Future Directions, *Educational Psychology Review*, Vol. 17, No. 2, pp. 147-177. 160.
- Perrenet, J.C., J.F. Groote & E.J.S. Kaasenbrood (2004). Denkniveaus bij algoritmen. *Tijdschrift voor informatica-onderwijs (TINFON)*, 13e jaargang, nr. 4, pp. 116-118.
- Perrenet, J.C., J.F. Groote & E.J.S. Kaasenbrood (2005a). Denkniveaus bij algoritmen; In: *NIOC proceedings (Nationaal Informatica Onderwijs Congres, Groningen, 2004)* pp. 90-95; Uitg. Passage, Groningen.
- Perrenet, J.C., J.F. Groote & E. Kaasenbrood (2005b). Exploring Students' Understanding of the Concept of Algorithm: Levels of Abstraction; In: *Proceedings of the 10th annual SIGCSE-conference on Innovation and technology in computer science education*, Caparica, Portugal pp. 64-68; ACM, New York.
- Perrenet, J.C. & E. Kaasenbrood (in print). Levels of Abstraction in Students' Understanding of the Concept of Algorithm: the Qualitative Perspective; *Paper, te presenteren op 11e jaarlijkse conferentie van de ITiCSE*, Bologna, 2006.
- Psychology of Programming Interest Group; <http://www.ppig.org/>
- Platform Academische Vorming: http://w3.tm.tue.nl/nl/capaciteitsgroepen/av/platform_academische_vorming/
- Skemp, R.S. (1976). Instrumental Understanding and Relational Understanding. *Mathematics Teaching*, 77, pp. 20-26.
- Special Interest Group fo Computer Science Education; <http://www.sigcse.org/>
- Tall, E. & Thomas, T. (Ed.) (2002). *Intelligence, Learning and Understanding in Mathematics; a tribute to Richard Skemp*. Post Pressed, Flaxton.

Auteurs

Dr.drs.Jacob C. Perrenet (j.c.perrenet@tue.nl) is aan de Technische Universiteit Eindhoven onder meer onderwijskundig medewerker bij de opleiding Technische Informatica, Prof.dr.ir. Jan Friso Groote (j.f.groote@tue.nl) is opleidingsdirecteur van de informaticaopleiding en Eric Kaasenbrood (e.j.s.kaasenbrood@student.tue.nl) is student informatica; het postadres is: Technische Universiteit Eindhoven, Faculteit Wiskunde en Informatica, Postbus 513, 5600 MB Eindhoven.