From the Bottom to the Top?

Betsy van Dijk
Herman Koppelman

In het informatica-onderwijs leggen we al tientallen jaren de nadruk op het systematisch, top-down ontwerpen van programmatuur en informatiesystemen. Een dergelijke aanpak zorgt immers bij het oplossen van omvangrijke problemen voor beheersing van de complexiteit en voor leesbaarheid en testbaarheid van de oplossing. Vooral het ontwerpproces vinden we belangrijk. Het leren van een programmeertaal of specificatietaal om het ontwerp te implementeren is weliswaar noodzakelijk, maar komt op de tweede plaats.

In het programmeeronderwijs zijn we overgestapt naar talen, zoals Pascal en Modula, die mede zijn ontwikkeld om een gestructureerde probleemaanpak te stimuleren en om modulair werken te vergemakkelijken. We hebben schematechnieken, zoals programmastructuurdiagrammen (PSD's) en dataflow-diagrammen (DFD's) ingevoerd om systematisch, top-down ontwerp af te dwingen, en ook de leerboeken leggen de nadruk op gestructureerd programmeren en gestructureerd systeemontwerp. Kortom, we hebben er de afgelopen decennia veel aan gedaan om studenten een systematische, top-down probleemaanpak bij te brengen. Toch lukt het, vooral in het inleidende onderwijs, maar mondjesmaat om studenten ertoe te bewegen een dergelijke probleemaanpak toe te passen. Waarom willen of kunnen studenten die gestructureerde aanpak niet van ons overnemen?

Waar gaat het mis?

Essentieel voor het oplossen van ontwerpproblemen is het onderscheiden van deelproblemen die afzonderlijk kunnen worden uitgewerkt, en de compositie van die deelproblemen tot een model voor de oplossing van het totale probleem. Als we de principes van een top-down probleemaanpak uitleggen en illustreren (vaak met een voorbeeld uit het dagelijks leven) lijken studenten die nog helemaal te begrijpen. Ook als we de aanpak demonstreren aan de hand van een voorbeeld kunnen ze ons nog aardig volgen. De moeilijkheden worden pas duidelijk als de studenten zelf aan het werk gaan. Dan blijkt het moeilijk te zijn de aanwijzingen uit de top-down probleemaanpak in praktijk te brengen.

Waarom gaat het mis?

Het aanleren van een top-down probleemaanpak wordt bemoeilijkt door een aantal factoren. Allereerst blijkt het opdelen van een probleem in deelproblemen erg moeilijk te zijn voor beginners. Voor het onderscheiden van geschikte deelproblemen is kennis van de bottom nodig. Ervaren programmeurs en systeemanalisten herkennen in een probleem al snel deelproblemen van een bekend type. Zonder die problemen meteen in detail op te lossen, 'weten' ze dat de uiteindelijke oplossing geen moeilijkheden zal opleveren. Het opdelen in deelproblemen wordt daardoor aardig ondersteund. Verder ligt uitstel van het in detail uitwerken van een deelprobleem ook wat meer voor de hand als je erop vertrouwt dat dit wel zal lukken.

Een tweede probleem dat wellicht (deels) een gevolg is van het vorige, is de antwoordgerichtheid van studenten; hun neiging onmiddellijk delen van de oplossing in detail uit te werken, zonder te bedenken hoe die delen passen in de totaaloplossing van het probleem. Deze werkwijze is hardnekkig. De maatregelen die we nemen in een poging top-down werken af te dwingen, leiden er vaak alleen toe dat studenten eerst 'gewoon' bottom-up (of beter onsystematisch) een oplossing ontwerpen en daarna proberen een top-down aanpak erbij te bedenken. Een nuttige oefening, als het tenminste lukt, maar de bedoeling is het niet, en de kans op mislukking is groot.

De derde factor betreft een dilemma dat inherent is aan inleidend onderwijs in een systematische probleemaanpak. Oefenproblemen voor de studenten moeten redelijk omvangrijk zijn om de voordelen van een top-down probleemaanpak te kunnen illustreren. Tegelijkertijd mogen oefenproblemen voor beginnende studenten weer niet te moeilijk zijn wegens het risico dat studenten vroegtijdig afhaken.

De tot nu toe genoemde factoren hadden alle min of meer betrekking op kenmerken van de beginnende student. Kort samengevat komt het er op neer, dat de gebrekkige kennis van beginnende studenten het leren van een top-down aanpak belemmert.

Tot slot nog een factor die betrekking heeft op de rol die wijzelf, de informaticadocenten, spelen bij dit didactische probleem. Het vinden van een geschikte decompositie van de problemen die we aan onze studenten voorleggen, is voor ons in het algemeen routinewerk. Maar kunnen we eigenlijk wel zo goed uitleggen waarom we een probleem op een bepaalde manier opsplitsen? Ook bij het beoordelen van oplossingen van studenten constateren we vaak in één oogopslag dat een bepaalde decompositie niet werkbaar is. Maar kunnen we ook aangeven waarom deze niet deugt? Of presenteren we maar liever meteen een beter alternatief? Als we hier tekortschieten, zullen de studenten ons niet zo snel corrigeren. Blij dat ze de voorbeelden

(passief) kunnen volgen, realiseren ze zich waarschijnlijk niet, dat gemaakte keuzen impliciet blijven en dat alternatieve oplossingen denkbaar zijn. En als blijkt dat zelf uitvoeren niet meevalt, wordt een gepresenteerde aanzet tot een oplossing dankbaar aanvaard.

Wat kunnen we hieraan doen?

Ontwerpen blijft moeilijk, zeker voor beginnende studenten. In het voorgaande is er al op gewezen dat de moeilijkheden deels voortkomen uit het ontbreken van kennis van de bottom bij beginnende studenten. Onder bottom moeten niet alleen de taalelementen worden verstaan, maar evengoed bouwstenen in de vorm van vaak voorkomende (deel)problemen waarvoor een standaardoplossing bekend is.

Er dient dus de nodige aandacht aan de bottom besteed te worden. Dat moet er niet toe leiden dat eerst de syntax en de semantiek van een taal uitvoerig worden bestudeerd voordat het ontwerpen aan bod komt. Voor studenten is het saai en weinig motiverend om eerst lang te moeten oefenen met taalelementen en kleine opdrachten. Precies om die reden is men in het onderwijs in vreemde talen immers ook afgestapt van het eerst jarenlang vooral oefenen met grammatica en woordjes.

Wegens de complexiteit van de te verwerven vaardigheden is een voor de hand liggend didactisch uitgangspunt van toepassing: we moeten niet teveel tegelijk willen. De te leren vaardigheden moeten met andere woorden gedoseerd worden aangeboden. Vandaar de suggestie om in het onderwijs afzonderlijk aandacht te besteden aan:

1. ontwerpen waarbij gebruik gemaakt wordt van een bestaande bottom;
2. ontwerpen waarbij ook de bouwstenen zelf moeten worden bedacht.

De eerste stap.

In de eerste stap gaat het er om de studenten te leren hoe ze in een ontwerp gebruik kunnen maken van al bestaande bouwstenen in de vorm van kant-en-klare 'bibliotheken' of modulen. Bij het programmeren gaat het daarbij voornamelijk om procedures en functies en, in objectgeoriënteerde talen, om objectklassen. Van deze bouwstenen hoeft alleen bekend te zijn wat het effect is, niet de wijze waarop dat effect wordt bereikt. Op deze wijze kunnen sneller omvangrijke en interessante problemen worden opgelost, te meer omdat al met een beperkt aantal bouwstenen vrij omvangrijke problemen kunnen worden aangepakt. Een ander voordeel is, dat vanaf het begin wordt geoefend met het onderscheid tussen specificaties en implementaties, ofwel tussen het wat en het hoe. Inzicht in dit onderscheid ligt ten grondslag aan top-down ontwerp, en is dan ook een voorwaarde om complexe problemen te kunnen oplossen.

De nadruk ligt in deze stap dus vooral op het ontwerp, de specificatie. Maar ook de effecten van de te gebruiken bouwstenen moeten worden bestudeerd. Deze aanpak kan daarom alleen slagen als deze bouwstenen goed gestructureerd zijn. De bibliotheken of modulen moeten zodanig zijn geordend, dat het beginners weinig tijd kost de inhoud te leren gebruiken. Vooral bij objectgeoriënteerde talen, zoals Smalltalk, is dit het geval. Dit kan een argument zijn om voor inleidend programmeeronderwijs te kiezen voor dergelijke talen.

De tweede stap.

In de tweede stap krijgen studenten problemen aangeboden die ze moeten opsplitsen in deelproblemen die zelf geen al bestaande bouwstenen zijn. We zouden ook kunnen zeggen: een bestaande bottom moet worden uitgebreid. Studenten moeten dus zelf bepalen welke bouwstenen nodig zijn, het effect daarvan beschrijven en ze, later, implementeren. Vooral in deze stap is het moeilijk voor studenten om de systematiek van het top-down ontwerpen toe te passen, ook al hebben ze daarmee in de eerste stap uitgebreid geoefend. Alleen demonstreren hoe een modeloplossing kan worden ontworpen, is voor veel studenten niet voldoende. Daarom een paar suggesties:

- Maak tijdens demonstraties de stappen van de probleemaanpak expliciet, met de redenen om zo'n aanpak te gebruiken.
- Maak bij het geven van voorbeelden zoveel mogelijk expliciet hoe je komt tot een bepaalde decompositie van het probleem.
- Presenteer ook alternatieve decomposities (waaronder ook onhandige) met voor- en nadelen.

De moeilijkheidsgraad van problemen kan verder worden beheerst door in eerste instantie problemen aan te bieden die kunnen worden opgelost door toevoeging van enkele voor de hand liggende bouwstenen. Pas later kunnen problemen aan bod komen waarvoor de zelf te ontwerpen bouwstenen talrijker en minder voor de hand liggend zijn.

In onze ijver om studenten een systematische, top-down probleemaanpak bij te brengen, hebben we in het verleden wellicht te weinig aandacht besteed aan de ontwikkeling van de bottom. Die vinden we immers minder belangrijk. Sterker nog, die moet zonder veel moeite kunnen worden ingewisseld voor een andere. Toch is er alle reden om te denken dat een bottom aanwezig moet zijn om een benadering vanuit de top mogelijk te maken.

Auteurs

Drs. E.M.A.G. van Dijk is werkzaam als universitair docent en vakdidacticus informatica aan de Universiteit Twente.

Ir. H. Koppelman is werkzaam als docent aan de Universiteit Twente en aan de Open universiteit.

Correspondentie-adres: Faculteit Informatica, Universiteit Twente, Postbus 217, 7500 AE, Enschede, telefoon, 053-893781.

Programmeren door **completeren**

Jeroen van Merriënboer

Het leren en onderwijzen van ontwerpactiviteiten neemt een belangrijke plaats in binnen het informatica-onderwijs. Veelal ligt in dit onderwijs het accent op het zelfstandig oplossen van ontwerpproblemen. Dit artikel beschrijft de 'completion strategy'. Dit is een trainingsstrategie waarbij niet het zelfstandig ontwerpen van complete systemen centraal staat, maar juist het aanvullen en uitbreiden van bestaande, goed-gestructureerde maar *onvolledige systemen*. Binnen inleidend programmeeronderwijs zijn met deze trainingsstrategie al uitstekende resultaten behaald.

1 Inleiding

Bij het leren ontwerpen van hard- en software-systemen worden diverse trainingsstrategieën toegepast om de leerprocessen te ondersteunen. Zo kan er binnen inleidend programmeeronderwijs gebruik gemaakt worden van strategieën waarbij men lerenden stap-voor-stap nieuwe elementaire commando's en semantische aspecten van een programmeertaal presenteert. Na elke stap worden dan opdrachten aangeboden om de nieuwe elementen te oefenen. Deze benadering noemt men wel de *spiral approach* (Shneiderman, 1977). Men treft hem aan in studieboeken met namen als *Learning BASIC Step by Step* of *An active Approach to Logo*. Daarnaast kan men gebruik maken van strategieën die een *topdown-ontwerpmodel* centraal stellen door lerenden programmeerproblemen systematisch uiteen te laten rafelen, coderen en testen. Deze benadering is afkomstig uit de beweging van het 'gestructureerd programmeren' (Wirth, 1974). Men treft hem aan in studieboeken met namen als *Problem Solving with Algol* of *An Introduction to Structured Design and Pascal*.

Er valt veel op te merken over de verschillen tussen dergelijke trainingsstrategieën (zie Van Merriënboer en Krammer, 1987). In dit artikel ligt de nadruk echter op twee treffende overeenkomsten. De eerste overeenkomst is dat doorgaans het door de lerende zelfstandig oplossen van problemen centraal staat, zonder hierbij een deel van de gewenste oplossing te presenteren. In de opdrachten gaat het om het ontwerpen van complete systemen en de complexiteit hiervan neemt toe naarmate het onderwijs vordert. De tweede overeenkomst is dat er gebruik gemaakt wordt van een tweetal duidelijk te onderscheiden instructieve componenten. Er is steeds sprake van (1) nieuwe informatie die, ter toelichting en illustratie, wordt gecombineerd met uitgewerkte voorbeelden en (2) problemen die zelfstandig door de lerende opgelost moeten worden.

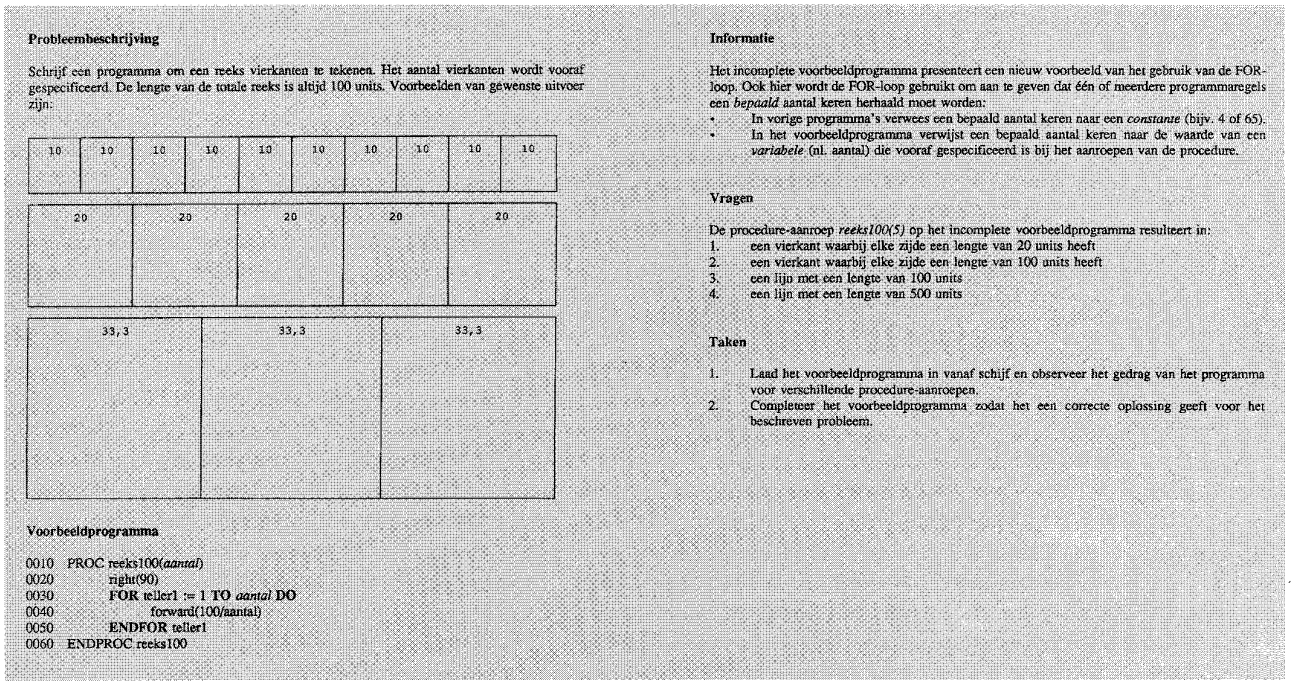
De *'completion strategy'* (Van Merriënboer en Krammer, 1990) biedt een alternatief voor conventionele trainingsstrategieën. In deze benadering staat niet het zelfstandig ontwerpen van steeds complexere, complete systemen centraal, maar juist het aanvullen en uitbreiden van steeds grotere gedeelten van onvolledige systemen. De te completeren systemen zijn goed gestructureerd en worden op een begrijpelijke wijze gepresenteerd. Tot op heden heeft onderzoek naar de effectiviteit van de 'completion strategy' zich toegespitst op inleidend onderwijs in het programmeren. De trainingsstrategie is echter gebaseerd op algemene cognitief-psychologische inzichten met betrekking tot het leren van ontwerpvaardigheden. Als zodanig is de trainingsstrategie ook geschikt voor het onderwijzen van andere ontwerpactiviteiten binnen de informatica.

De structuur van dit artikel is als volgt. De tweede paragraaf geeft een beschrijving van 'completeeropdrachten': deze kunnen gezien worden als de bouwstenen van de 'completion strategy'. De paragrafen 3 en 4 gaan achtereenvolgens in op het gebruik van uitgewerkte voorbeelden en op de hieruit voortvloeiende voordelen van de 'completion strategy' ten opzichte van andere trainingsstrategieën. De vijfde paragraaf beschrijft een tweetal studies naar de effectiviteit van de 'completion strategy' in klassikaal onderwijs en computerondersteund onderwijs (COO). De laatste paragraaf bevat enige aanbevelingen en een korte beschrijving van lopend onderzoek.

2 Completeeropdrachten als bouwstenen

In tegenstelling tot conventionele trainingsstrategieën maakt de 'completion strategy' gebruik van slechts één instructieve component, de completeeropdracht. Figuur 1 geeft een vereenvoudigd voorbeeld van zo'n completeeropdracht. Zoals aangegeven in deze figuur kan een completeeropdracht bestaan uit ten hoogste vijf elementen:

1. *Probleembeschrijving:* een duidelijke beschrijving van een programmeerprobleem in natuurlijke taal. Dit is het enige element dat voor alle completeeropdrachten aanwezig moet zijn (tezamen met ten minste één ander element).
2. *Voorbeeldprogramma:* één meer of minder uitgewerkte oplossing voor het probleem zoals beschreven onder punt 1, in de vorm van een goed leesbaar, goed gestructureerd maar incompleet programma. Het voorbeeldprogramma kan ook afwezig zijn of juist compleet zijn: deze situaties worden opgevat als de extremen van een continuüm.
3. *Informatie:* uitleg over nieuwe aspecten van het programmeren of de programmeertaal. Deze nieuwe aspecten worden altijd geïllustreerd in het voorbeeldprogramma zoals beschreven onder punt 2.
4. *Vragen:* meerkeuzevragen of open vragen over bekend veronderstelde aspecten van de programmeertaal of het programmeren, wederom naar aanleiding van hun toe-



Probleembeschrijving

Schrijf een programma om een reeks vierkanten te tekenen. Het aantal vierkanten wordt vooraf gespecificeerd. De lengte van de totale reeks is altijd 100 units. Voorbeelden van gewenste uitvoer zijn:

Voorbeeldprogramma

```
0010  PROC reeks100(aantal)
0020        right(90)
0030        FOR teller1 := 1 TO aantal DO
0040              forward(100/aantal)
0050        ENDFOR teller1
0060  ENDPROC reeks100
```

Informatie

Het incomplete voorbeeldprogramma presenteert een nieuw voorbeeld van het gebruik van de FOR-loop. Ook hier wordt de FOR-loop gebruikt om aan te geven dat één of meerdere programmaregels een *bepaald* aantal keren herhaald moet worden:

- In vorige programma's verwees een bepaald aantal keren naar een *constante* (bijv. 4 of 65).
- In het voorbeeldprogramma verwijst een bepaald aantal keren naar de waarde van een *variabele* (nl. aantal) die vooraf gespecificeerd is bij het aanroepen van de procedure.

Vragen

De procedure-aanroep *reeks100(5)* op het incomplete voorbeeldprogramma resulteert in:

1. een vierkant waarbij elke zijde een lengte van 20 units heeft
2. een vierkant waarbij elke zijde een lengte van 100 units heeft
3. een lijn met een lengte van 100 units
4. een lijn met een lengte van 500 units

Taken

1. Laad het voorbeeldprogramma in vanaf schijf en observeer het gedrag van het programma voor verschillende procedure-aanroepen.
2. Completeer het voorbeeldprogramma zodat het een correcte oplossing geeft voor het beschreven probleem.

Figuur 1 Een vereenvoudigd voorbeeld van een completeeropdracht waarin gebruik gemaakt wordt van de programmeertaal COMAL-80

passing in het voorbeeldprogramma zoals beschreven onder punt 2.

5. *Taken:* een beschrijving van de taak of taken die de lerende moet verrichten. Meestal betreft de taak het completeren van het onvolledige voorbeeldprogramma zoals beschreven onder punt 2. Maar ook het schrijven van een programma (als er géén voorbeeldprogramma beschikbaar is) of het veranderen of aanvullen van een programma (als het voorbeeldprogramma reeds volledig is) behoren tot de mogelijkheden.

3 Voorbeeldgebruik en de verwerving van templates

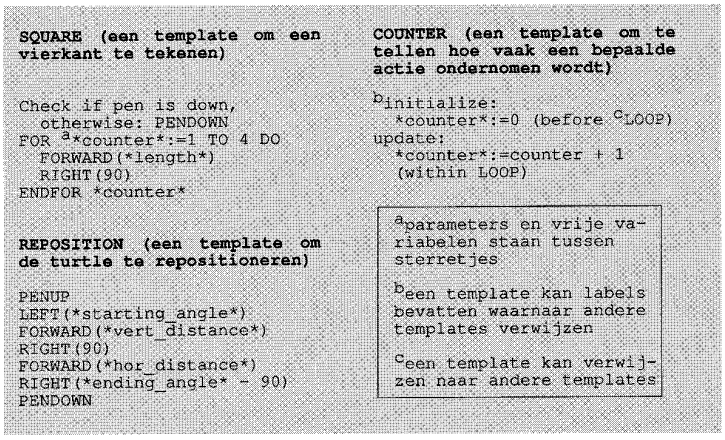
In het algemeen vormen uitgewerkte voorbeelden een uitstekend middel om nieuwe informatie te illustreren en te concretiseren. Daarnaast bieden zij de mogelijkheid om abstracte kennis die door een docent moeilijk te verwoorden is op een heel natuurlijke wijze over te dragen aan de lerenden. Voorbeelden in de vorm van oplossingen voor gespecificeerde problemen spelen een essentiële rol bij het leren van een ontwerpvaardigheid zoals programmeren (van Merriënboer & Paas, 1990). Lerenden kunnen vanuit concrete voorbeeldprogramma's abstraheren en zo 'veralgemeende' kennis verwerven omtrent principes van het programmeren, bruikbare ontwerptechnieken en bestaande standaardoplossingen voor veel voorkomende (deel)problemen. In de psychologie worden zulke abstracte kenniselementen doorgaans cognitieve schemata genoemd. Deze schemata geven, nadat zij geleerd zijn, richting aan het oplossen van nieuwe problemen.

Belangrijke cognitieve schemata die bij het leren programmeren verworven moeten worden noemt men wel *templates* (Linn, 1985), 'programming plans' (Soloway, 1985) of 'beacons' (Wiedenbeck, 1986). Templates kunnen ruwweg opgevat worden als stereotiepe patronen voor programmacode die geassocieerd zijn met bepaalde deelproblemen en/of programmeerdoelen. Zij zijn grotendeels onafhankelijk van de programmeertaal die gebruikt wordt (voor zover deze talen tot dezelfde familie behoren, bij voorbeeld imperatieve talen, logische talen of functionele talen) en zij vormen tezamen een hiërarchie van veralgemeniseerde kennis. Templates hoog in de hiërarchie worden bij voorbeeld gebruikt om een onderscheid te maken tussen invoer, verwerking en uitvoer; templates laag in de hiërarchie worden gebruikt om elementaire commando's, volgens de juiste syntactische regels, met hun argumenten te combineren tot zinvolle statements. In figuur 2 wordt een drietal templates weergegeven in een niet-formele notatie.

Om een hiërarchie van templates te kunnen ontwikkelen moeten lerenden tijdens het onderwijs geconfronteerd worden met een groot aantal programmeerproblemen èn hun gerelateerde oplossingen in de vorm van goed-gestructureerde programma's. De beschikbaarheid van een vocabulair van templates ondersteunt vervolgens het effectief ontwerpen van programma's, vergroot de kans dat semantisch correcte oplossingen bereikt worden en is tevens behulpzaam bij het correct coderen van deze oplossingen.

4 Enige voordelen van de 'completion strategy'

De belangrijkste voordelen van de 'completion strategy' liggen in de koppeling tussen voorbeeld en oefening: het incomplete programma dient tegelijkertijd als uitgewerkt voorbeeld, dat grondig bestudeerd moet worden, en als oefening, die gecompleteerd moet worden. Een eerste voordeel heeft daarom te maken met het stimuleren van *bewuste abstractie* ('mindful abstraction'; Salomon & Perkins, 1987). Bij traditionele trainingsstrategieën is er voor de lerenden meestal geen dwingende noodzaak om uitgewerkte voorbeelden goed te bestuderen. Dit bestuderen en bewust abstraheren is een proces dat moeite en mentale inspanning kost (Anderson, 1987), zodat lerenden eventuele voorbeelden veelal slechts vluchtig zullen bekijken en zo snel mogelijk met oefenen beginnen. Bij de 'completion strategy' worden lerenden gedwongen om de incomplete voorbeelden zorgvuldig te bestuderen. Als zij dit niet doen, zullen zij



SQUARE (een template om een vierkant te tekenen)

```
Check if pen is down,
  otherwise: PENDOWN
FOR [a]*counter*:=1 TO 4 DO
  FORWARD(*length*)
  RIGHT(90)
ENDFOR *counter*
```

REPOSITION (een template om de turtle te repositioneren)

```
PENUP
LEFT(*starting_angle*)
FORWARD(*vert_distance*)
RIGHT(90)
FORWARD(*hor_distance*)
RIGHT(*ending_angle* - 90)
PENDOWN
```

COUNTER (een template om te tellen hoe vaak een bepaalde actie ondernomen wordt)

```
[b]initialize:
  *counter*:=0 (before [c]LOOP)
update:
  *counter*:=counter + 1
  (within LOOP)
```

[a]parameters en vrije variabelen staan tussen sterretjes

[b]een template kan labels bevatten waarnaar andere templates verwijzen

[c]een template kan verwijzen naar andere templates

Figuur 2 Een niet-formele beschrijving van een drietal eenvoudige templates

immers niet in staat zijn om het incomplete programma correct aan te vullen.

Een tweede voordeel betreft de *directe beschikbaarheid* van uitgewerkte voorbeelden tijdens het uitvoeren van de programmeertaken. Zoals hierboven vermeld zullen lerenden binnen traditionele trainingsstrategieën de uitgewerkte voorbeelden aanvankelijk slechts vluchtig bekijken. Zij gaan pas gericht op zoek naar bruikbare voorbeelden als zij bij het oplossen van een programmeerprobleem in een impasse raken. Het zoeken van bruikbare informatie is dan echter een moeilijke opgave. Er moet gebladerd worden in lesboeken zonder dat men weet of het 'gezochte' beschikbaar is. Tevens loopt men het risico te vergeten wat er gezocht wordt nog voordat dit gevonden is; men moet immers ook nog allerlei andere zaken die te maken hebben met het probleemoplosproces in het geheugen 'actief' houden. Bij de 'completion strategy' hebben de lerenden alle voorafgaande uitgewerkte (incomplete) voorbeelden grondig bestudeerd. Zij weten dus beter welke informatie beschikbaar is en waar deze gevonden kan worden. Bovendien kan een docent er eenvoudig voor zorgen dat bruikbare informatie onmiddellijk in de te completeren oplossing beschikbaar is. Als de lerende bij voorbeeld voor de eerste keer gebruik moet maken van een bepaalde herhalingsstructuur (zoals REPEAT ... UNTIL), kan men een incompleet programma aanbieden waar deze structuur reeds deel van uitmaakt èn waarbij deze structuur nodig is om het programma correct te completeren.

Een derde voordeel van de 'completion strategy' betreft de *mentale verwerkingsbelasting* die van de lerenden vereist wordt. Deze verwerkingsbelasting is extreem hoog bij het oplossen van conventionele programmeerproblemen en kan veel van de gemaakte programmeerfouten verklaren (Anderson & Jeffries, 1985). Ook verklaart een hoge verwerkingsbelasting waarom het, binnen conventionele trainingsstrategieën, zo lastig is om tijdens het oplossen van programmeerproblemen ook nog op zoek te gaan naar bruikbare informatie en voorbeelden. Bij de 'completion strategy' is de verwerkingsbelasting die vereist is om de programmeertaak tot een goed einde te brengen lager omdat een deel van de oplossing al beschikbaar is. Een positief gevolg hiervan is dat de lerenden de beschikking hebben over

voor bewuste abstractie vanuit aangeboden voorbeelden. Dit proces komt de verwerving van templates ten goede.

Samenvattend kan men stellen dat binnen traditionele benaderingen voorbeelden gepresenteerd worden zonder de noodzaak om deze zorgvuldig te bestuderen, dat oefeningen gepaard gaan met een hoge verwerkingsbelasting, en dat bruikbare informatie niet direct beschikbaar is tijdens het oplossen van programmeerproblemen. Binnen de 'completion strategy' presenteert men (incomplete) voorbeelden die zorgvuldig bestudeerd móeten worden, oefeningen die gepaard gaan met een lagere verwerkingsbelasting, en informatie die direct beschikbaar is tijdens oefening. In combinatie leiden deze factoren tot een meer produktief leerproces dat de verwerving van templates ten goede komt. Een bijkomend heel ander voordeel is dat het leren plaats vindt in een omgeving die grote overeenkomsten vertoont met een professionele programmeeromgeving. Ook daar ziet men immers meer en meer 'reverse engineering' benaderingen, waarbij nieuwe software ontwikkeld wordt op basis van bibliotheken met procedures en bestaande, aan te vullen of aan te passen modules.

5 Uitgevoerd onderzoek naar de 'completion strategy'

Een tweetal empirische studies geeft inzicht in de bereikte leeruitkomsten bij toepassing van de 'completion strategy' in klassikaal onderwijs (van Merriënboer, 1990a, 1990b) en computerondersteund onderwijs (van Merriënboer, 1990a; van Merriënboer & de Croock, in druk). De klassikale cursus werd verzorgd in een computerlokaal en omvatte 10 lessen. Leerlingen uit de vierde en vijfde klas HAVO/VWO met een gemiddelde leeftijd van 16 jaar namen aan de cursus deel. De COO-cursus bestond uit een computerondersteunde instructie van ongeveer drie uur. De instructie werd gevolgd door eerstejaars studenten uit het WO met een gemiddelde leeftijd van 19 jaar. In beide studies was het cursusdoel het verwerven van enige elementaire programmeervaardigheden met betrekking tot 'turtle graphics' in COMAL-80 (Christensen, 1982). Bij aanvang hadden de lerenden geen ervaring met programmeren.

In elk van de uitgevoerde experimenten ontving de helft van de totale groep (genereergroep) een traditionele vorm van instructie en de andere helft (completeergroep) instructie volgens de 'completion strategy'. De genereergroep ontving afwisselend (a) informatie over programmeren en de programmeertaal die werd geïllustreerd in complete voorbeeldprogramma's en (b) conventionele programmeerproblemen waarvoor zelfstandig nieuwe programma's ontworpen en gecodeerd moesten worden. De completeergroep ontving slechts completeeropdrachten. De informatie die deel uitmaakte van deze completeeropdrachten was identiek aan de informatie die de genereergroep ontving, maar werd geïllustreerd aan de hand van de incomplete voorbeeldprogramma's. Bij het klassikaal onderwijs werd gebruik gemaakt van werkboeken die door de leerlingen zelfstandig doorgewerkt konden worden. Bij het computerondersteund onderwijs werd gebruik gemaakt van een COO-programma waarvan de opbouw wordt weergegeven in figuur 3.

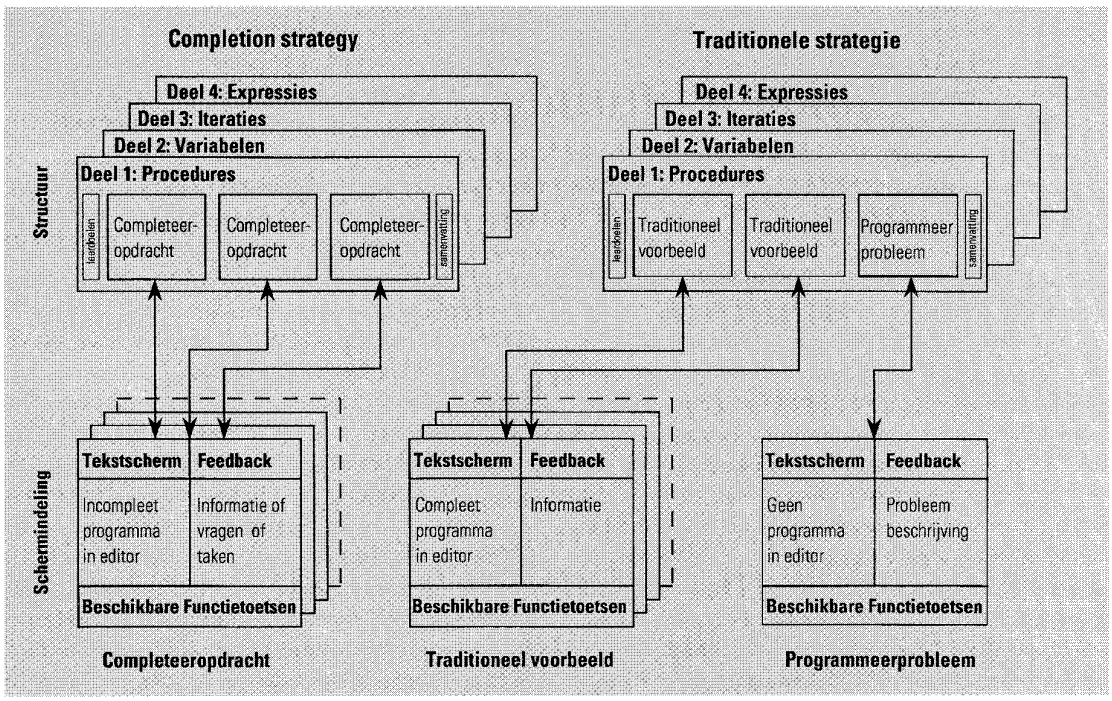
In beide studies werden de leeruitkomsten na afloop van de cursus vastgesteld. Hiertoe werd gebruik gemaakt van een programmaconstructietest alsmede een tweetal toetsen bestaande uit meerkeuzevragen. De programmaconstructietest bestond uit conventionele programmeerproblemen waarvoor leerlingen op papier een oplossing moesten uitwerken. De te schrijven programma's vereisten het gebruik van een aantal noodzakelijke templates. Twee observatoren scoorden het aantal correct gebruikte templates (interbeoordelaarsbetrouwbaarheid r_{obs}= .98 voor de klassikale studie en .94 voor de COO-studie) en, op een 5-puntsschaal, de semantische correctheid van het programma (r_{obs} = .86 respectievelijk .92). Tevens werd het percentage syntactisch correct gecodeerde programmaregels vastgesteld. De eerste meerkeuzetoets bevatte vragen om de passieve kennis van elementaire commando's, correcte statements en syntactische regels van de programmeertaal vast te stellen; de tweede toets bevatte vragen om begrip van de werking van bestaande, goed leesbare en goed gestructureerde programma's te meten.

Tabel 1 geeft een overzicht van de leeruitkomsten. De tabel laat zien dat leerlingen uit de completeergroep in beide studies op de programmaconstructietest hoger scoorden dan leerlingen uit de genereergroep. Het aantal correct gebruikte templates was groter, de programma's waren vaker semantisch correct, en het percentage syntactisch correct gecodeerde regels was hoger. Deze bevindingen steunen de idee dat de 'completion strategy' positieve effecten heeft op de verwerving van templates. Enerzijds rechtstreeks, omdat meer templates correct gebruikt worden, maar ook indirect, omdat de beschikbaarheid van templates verklaart waarom lerenden beter in staat zijn om semantisch en syntactisch correcte (deel)oplossingen te bereiken. Voor de overige leeruitkomsten werden geen significante verschillen gevonden – met uitzondering van 'taalkennis' bij de COO-studie. Het ontbreken van verschillen bij deze leeruitkomsten is waarschijnlijk te verklaren uit het feit dat de beschikbaarheid van templates hierbij geen (grote) rol speelt.

Binnen de COO-studie werden tevens de leeractiviteiten gevolgd. De completeergroep en de genereergroep besteedden even veel tijd aan het doorlopen van het gehele COO-programma, maar de completeergroep besteedde relatief méér tijd aan oefenen. Dit is als volgt te verklaren. Studenten uit de genereergroep besteedden een aanzienlijk deel van hun tijd aan het *opnieuw* bestuderen van reeds eerder bestudeerde informatie (gemiddeld 9,5% van de totale studieduur). Hiertoe gingen zij terug naar eerdere 'schermen' tijdens het schrijven van hun programma's. Studenten uit de completeergroep besteedden daarentegen weinig tijd aan het opnieuw bestuderen van al eerder gepresenteerde informatie (slechts 0,8% van de totale studieduur). Zij gingen slechts zelden terug naar eerdere schermen tijdens het completeren van hun programma's. Hiermee in overeenstemming is het feit dat studenten uit de genereergroep bij het bestuderen van informatie en voorbeeldprogramma's veel aantekeningen over de elementaire commando's en de syntax van de programmeertaal maakten (gemiddeld 65 regels). Deze aantekeningen gebruikten zij vervolgens tijdens het schrijven van hun programma's. Studenten uit de completeergroep maakten minder aantekeningen (gemiddeld 35 regels).

6 Conclusies

De verkregen resultaten ondersteunen de idee dat completeeropdrachten zoals gebruikt in de 'completion strategy' verschillende belangrijke voordelen bieden. Ten eerste worden leerlingen gestimuleerd om de geboden informatie en (incomplete) voorbeeldprogramma's zorgvuldig te bestuderen. Deze informatie hoeft niet voorafgaand aan de programmeertaak onthouden te worden (of: opgeschreven in de



Figuur 3 Weergave van de structuur en schermopbouw van de COO-programma's waarbinnen de 'completion strategy' en een conventionele strategie zijn vormgegeven

Tabel 1 Overzicht van de belangrijkste leeruitkomsten bij klassikaal onderwijs (N = 30) en computerondersteund onderwijs (N = 40)

Leeruitkomst	*Maximale score*	*Genereren*	*Completeren*
Klassikaal Onderwijs[a]			
Programmaconstructie:			
■ Templategebruik	24	10	15***
■ Semantische score	10	5	6*
■ Syntactische score (%)	100	76	86*
Meerkeuzetoetsen:			
■ Taalkennis	18	10	7
■ Programmabegrip	18	8	8
Computerondersteund Onderwijs[b]			
Programmaconstructie:			
■ Templategebruik	8	5,5	6,9**
■ Semantische score	5	2,5	3,1*
■ Syntactische score (%)	100	92,9	95,6
Meerkeuzetoetsen:			
■ Taalkennis	12	10,0	11,0**
■ Programmabegrip	12	8,9	9,4

[a] In deze studie werd gebruik gemaakt van een 'matched groups design'; de tabel presenteert medianen en toetsing heeft plaats gevonden met Wilcoxon's signed-rank test.

[b] In deze studie werd gebruik gemaakt van een 'randomized groups design'; de tabel presenteert gemiddelden en toetsing heeft plaats gevonden met Student's T-test.

*p < .05; **p < .025; ***p < .01.

vorm van aantekeningen) of tijdens de programmeertaak opgezocht te worden. In combinatie met de lagere verwerkingsbelasting heeft deze integratie van informatiebronnen (het probleem dat opgelost moet worden en de informatie die nodig is om het probleem op te lossen) een positief effect op het leerproces, omdat er meer 'aandacht' besteed kan worden aan bewuste abstractie vanuit de beschikbare voorbeelden. Kortom, het stimuleren van bewuste abstractie, de directe beschikbaarheid van informatie en de lage verwerkingsbelasting verklaren de meer produktieve leeractiviteiten, de betere verwerving van templates en de hogere leeruitkomsten die na de cursus behaald worden bij programmaconstructietaken.

7 Aanbevelingen en lopend onderzoek

Er is sprake van een interessante paradox: het oplossen van conventionele programmeerproblemen lijkt, in ieder geval voor beginners, niet de beste manier om programmeerproblemen te *leren* oplossen! De 'completion strategy' verdient de voorkeur boven trainingsstrategieën waarbij het oplossen van conventionele problemen centraal staat. Als men desalniettemin toch gebruik maakt van conventionele programmeerproblemen, lijkt het ongewenst om informatie en voorbeeldprogramma's die de lerenden kunnen helpen bij het oplossen van deze problemen afzonderlijk te presenteren (bij voorbeeld in aparte paragrafen van een lesboek of mondeling voorafgaand aan het oefenen). Het is beter om deze informatie expliciet te koppelen aan het voorliggende programmeerprobleem, zodat de beschikbaarheid van deze informatie tijdens het uitvoeren van de programmeeropdracht optimaal is.

Lopend onderzoek naar de 'completion strategy' ontwikkelt zich langs twee lijnen. Een eerste onderzoekslijn richt zich op de toepassing in intelligente tutorsystemen (ITS) voor programmeeronderwijs. Zo'n ITS genereert instructie die optimaal is toegesneden op de behoeften van een individuele lerende. Het onderzoek richt zich op de ontwikkeling van een ITS-module die dynamisch nieuwe completeeropdrachten genereert. De 'Completion Assignment Constructor' die momenteel geïmplementeerd wordt (CASCO; Krammer, van Merriënboer, en Maaswinkel, in druk) is derhalve op te vatten als een geautomatiseerd systeem dat op elk gewenst moment een completeeropdracht kan construeren, die precies is toegesneden op wat een individuele lerende reeds beheerst, nog niet beheerst en nog niet correct beheerst.

Een tweede onderzoekslijn betreft een uitbreiding van de toepassing van de 'completion strategy' naar het onderwijzen van andere ontwerpactiviteiten dan programmeren (zie bij voorbeeld Paas & van Merriënboer, 1992). Veel ontwerptaken die zich goed lenen voor het gebruik van de 'completion strategy' treft men aan binnen de informatica, waarbij men niet alleen moet denken aan het ontwerpen van programmatuur, maar ook aan het ontwerpen van informatiesystemen, kennissystemen, digitale systemen, enzovoorts. Maar ook de toepassing bij het onderwijzen van ontwerpactiviteiten in andere domeinen lijkt goed mogelijk, zoals het plannen van produktieprocessen of het schrijven van rapporten.

Literatuur

Anderson, J.R. (1987), 'Skill Acquisition: Compilation of Weak-method Problem Solutions', *Psychological Review*, 94, 192-210.

Anderson, J.R., en R. Jeffries (1985), 'Novice LISP Errors: Undetected losses of Information from Working Memory', *Human-Computer Interaction*, 1, 107-131.

Christensen, B.R. (1982), Beginning COMAL. Chichester, UK: Ellis Horwood.

Krammer, H.P.M., J.J.G. van Merriënboer, en R.M. Maaswinkel (in druk), 'Plan-based Delivery Composition in Intelligent Tutoring Systems for Introductory Computer Programming', in J. J. G. van Merriënboer (Red.), *Dutch Research on Knowledge-based Instructional Systems*, (Special Issue) *Computers in Human Behavior*.

Linn, M.C. (1985), 'The Cognitive Consequences of Programming Instruction in Classrooms', *Educational Researcher*, 14(5), 14-29.

Paas, F.G.W.C., en J.J.G. van Merriënboer (1992), 'Training voor transfer van statistische vaardigheden: Toepassing van een vier-componenten instructie-ontwerpmodel', in P.R.J. Simons en L. Verschaffel (Red.), *Themanummer Transfer. Tijdschrift voor Onderwijsresearch*, 17, 17-27.

Salomon, G., en D.N. Perkins (1987), 'Transfer of Cognitive Skills from Programming: When and how?', *Journal of Educational Computing Research*, 3, 149-169.

Shneiderman, B. (1977), 'Teaching Programming: A Spiral Approach to Syntax and Semantics', *Computers and Education*, 1, 193-197.

Soloway, E. (1985), 'From Problems to Programs via Plans: The Content and Structure of Knowledge for Introductory LISP Programming', *Journal of Educational Computing Research*, 1, 157-172.

Van Merriënboer, J.J.G. (1990a), Teaching Introductory Computer Programming – A Perspective from Instructional Technology, (ook verschenen als academisch proefschrift). Enschede, Bijlstra & Van Merriënboer.

Van Merriënboer, J.J.G. (1990b), 'Strategies for Programming Instruction in high School: Program Completion vs. Program Generation', *Journal of Educational Computing Research*, 6, 265-285.

Van Merriënboer, J.J.G., en M.B.M. de Croock (in druk), 'Strategies for Computer-based Programming Instruction: Program Completion vs. Program Generation', *Journal of Educational Computing Research*, 8(3).

Van Merriënboer, J.J.G., en H.P.M. Krammer (1987), 'Instructional Strategies and Tactics for the Design of Introductory Computer Programming Courses in high School', *Instructional Science*, 16, 251-285.

Van Merriënboer, J.J.G., en H.P.M. Krammer (1990), 'The 'Completion Strategy' in *Programming Instruction:* Theoretical and Empirical Support', in S. Dijkstra, B.H.M. van Hout Wolters en P.C. van der Sijde (Red.), *Research on Instruction: Design and Effects* (pp 45-61), Educational Technology Publications, Englewood Cliffs, NJ.

Van Merriënboer, J.J.G., H.P.M. Krammer en R.M. Maaswinkel (in druk), 'Automating the Planning and Construction of Programming Assignments for Teaching Introductory Computer Programming', in R.D. Tennyson, M. Spector en D. Muraida (Red.), *Automating Instructional Design, Development, and Delivery*, Springer Verlag, Berlijn.

Van Merriënboer, J.J.G., en F.G.W.C. Paas (1990), 'Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice', *Computers in Human Behavior*, 6, 273-289.

Wiedenbeck, S. (1986), 'Beacons in Computer Program Comprehension', *International Journal of Man-Machine Studies*, 25, 697-709.

Wirth, N. (1974), 'On the Composition of well-structured Programs', *Computing Surveys*, 6, 247-259.

Auteur

Dr. Jeroen J. G. van Merriënboer is werkzaam als universitair hoofddocent bij de vakgroep Instructietechnologie van de Faculteit der Toegepaste Onderwijskunde aan de Universiteit Twente. Adres: Universiteit Twente, Faculteit der Toegepaste Onderwijskunde, Postbus 217, 7500 AE Enschede.

Visual programming with Java; an alternative approach to introductory programming

Frank Wester*, Marleen Sint* and Peter Kluit**
* Faculty of Technical Sciences, Open Universiteit
PO Box 2960, 6401 DL Heerlen, the Netherlands
{frank.wester, marleen.sint}@ouh.nl
** Faculty of Technical Mathematics and Informatics, Delft University of Technology
PO Box 356, 2600 AJ Delft, the Netherlands
p.g.kluit@twi.tudelft.nl

Abstract

The appearance of the programming language Java and visual programming environments based on this language give new opportunities to teach introductory programming to university students. The authors are working on a new set of programming courses starting with Visual Programming. In this course the possibilities of visual programming environments are used to teach programming in an application oriented way with more emphasis on building user interfaces and using standard class libraries. Less attention is paid to algorithms. Because the applications in the course are of general interest, the course is suited for a broader audience than computer science students only.

1 Introduction

Of late, we have been rethinking our approach to introductory programming for our university students. Until now our introductory programming courses were based on the conventional procedural programming paradigm. They emphasized programming language constructs and concepts, good programming style, algorithm development and correctness. All these things are very important and we definitely want to go on teaching them, but in themselves they are insufficient to convey to our students that programming is an important skill and moreover can be fun.

This problem will be aggravated by our planned change-over to the object oriented paradigm and Java (as by now this is a common choice, we will not defend it here). OO programming is more complicated than straightforward procedural programming, whereas the benefits of OO are apparent only in larger programs. So in an introductory course, students may have to do a lot of work creating almost trivial programs. If we also take into account the enormous gap between programs developed in introductory courses and the standard software used by the students on their own computers, we cannot be amazed that freshmen students are not excited anymore about programming.

ITiCSE '97 Uppsala, Sweden

The recent introduction of visual development environments based on Java offers a possible solution to these problems. The availability of these environments, combined with the power of the standard Java packages suggest that students will be able to write grown-up applications with grown-up user interfaces even in a first course. In this way, they will almost immediately be able to appreciate the benefits of OO, and probably programming will be fun again. Another blessing we expect from this approach is earlier appreciation of program qualities like programming style and clear design.

However, it is not a trivial matter how to structure such a course if we want to teach students a balanced mix of (object oriented) programming techniques on the one hand, while on the other hand having them create user interfaces right from the start using a state of the art development environment.

Altogether, we will develop three new programming and algorithms courses based on Java, to be used in two universities. Here we focus on the first course, called Visual Programming, which will be used for CS students as well as students in other fields. The other two courses will be on Composition of OO-programs and on Algorithms and Data Structures and will be specifically meant for CS students. The Visual Programming course will be first taught in the fall semester of '97.

Characteristics of the Visual Programming course are:

- emphasis on creating a complete application including a user friendly interface rather than on just learning algorithms
- using a visual programming tool right from the start
- introducing object oriented principles in applications that use only standard Java classes
- separating functionality from interface aspects as early as possible.

Each of the characteristics will be elaborated in the following paragraphs.

2 Application directed versus algorithm directed

During the course, the students will develop a number of different applications. These applications are chosen in such a way, that they can be started small and will then be extended step by step when new language constructs are introduced. As projects, we will use, among others, small database applications, simulations and simple games. We will not use standard examples like calculating prime numbers, sorting or tree traversal. Computer science students, should, of course, become thoroughly familiar with all kind of standard algorithms, but these will be relegated to the course on Algorithms and Data Structures. In this way, the introductory course will also be of interest to students from other disciplines.

3 Using a visual programming tool

Inspired by succesful Rapid Application Development (RAD) tools like Visual Basic and Delphi (based on Object Pascal) there are several Java based products appearing right now (e.g. Visual Cafe (Symantec) or Open JBuilder (Borland)) that have similar (and more advanced) features. On top of the common features of any modern development environment, like good project management, syntax directed editing and good debugging facilities, these RAD tools offer the following features that may help us to accomplish the goals stated above.

– A visual form designer allows the programmer (the student) to easily create an interface for an applet or application. A tool palette contains widgets for creating simple interfaces (buttons, labels, edit and list boxes, choices, panels, ...); a property editor allows setting the initial values of the widgets' attributes in a simple way. Also, event handlers for the main events received by these widgets can easily be added. Java code for these widgets (declaration, creation and initialisation) and their event handling is automatically generated. Because of these tools, the students can use these widgets before they have a thorough knowledge of the language and the abstract window toolkit (awt). Moreover, the Java awt classes can be used to illustrate all kinds of object oriented concepts at an early moment in the course.

– These programming environments contain easy to use documentation to all the main standard Java packages (the application programmer interface or API); the students can find useful public methods of all the standard Java classes and apply them in their projects.

4 Using Java classes prior to making Java classes

The use of RAD tools allows fast development of non-trivial applications, but it also raises some serious didactical problems. The automatic code generation causes a huge 'blurb' of code in the program which at first sight a student will not be able to understand at all. Using Java in itself already raises these problems - even the most trivial Java program uses constructs the student cannot fully appreciate at first sight. Many Java books (e.g. [1, 2]) start with a "hello world" application, but have to gloss over all the issues related to "public static void main(String[ ] args)". Almost all even quite small applications need (static) methods of standard classes like Math, Integer and String. In our opinion it is admissible to sometimes use a construct while deferring full explanation to a later chapter, but this should not be done too often. There is a risk of bewildering the students, and we feel that the only Java book now available for novice programmers [3] has not sufficiently avoided this risk.

So we decided on a non-standard approach, somewhat comparable to the one proposed in [4]: we start with the introduction of basic OO-concepts like classes, instances, attributes and methods. We illustrate these concepts using a few standard classes (Button, Label, TextField, ListBox, Panel) from the awt package and a few other simple classes (e.g. String). We let the students build some simple applications that just require the definition of event handlers using nothing but calls to standard methods: no if-statements, no expressions beyond maybe a simple addition or multiplication, no loops etc. The students have to figure out themselves what methods to use. At this stage they do not yet use the standard API-documentation; instead, we give them access to a small, tailor-made version. After this first part of the course, they should, on top of the basic OO-concepts mentioned above, understand the difference between class and instance variables and methods (which again will be introduced using a suitable example) and have a basic notion of event handling. This entails that they should be able to understand what is going on in the code generated by the RAD environment (even though they might not yet be able to write such code themselves).

5 Separation of modeling and interface

We then continue with the more conventional programming constructs. In this part of the course we stress a clear separation between modeling and user interface, which means that students write separate classes modeling the functional aspects of the application under development. This part will be more conventional than the first one: we will no longer focus so much on standard classes and interface issues. We will be able to use them though, whenever necessary and without the need to gloss over any of the issues involved.

6 Conclusions

Compared to our former introductory course on programming the students that will have finished the Visual Programming course will probably lack some algorithmic skills (e.g. we won't do recursion anymore in this first course) but they will have gained a much better skill in completing 'real world applications' and also will have a better understanding of object oriented concepts. And we think they will have a lot more fun in programming.

References

1 Arnold, K. and Gosling, J., *The Java(TM) Programming Language*, Addison Wesley, 1996.

2 Campione, M. and Walrath, K., *The Java(TM) Tutorial*, Addison Wesley, 1996.

3 Deitel, H.M. and Deitel, P.J., *Java(TM) How to Program*, Prentice Hall, 1996.

4 Meyer, B., Towards an Object-Oriented Curriculum, *TOOLS 11*, Santa Barbara, August 1993; Prentice Hall 1993, pp. 585-594.

Checklist leermiddelen: aandacht voor deelonderwerpen

Methode: …….

Waar legt het boek de nadruk op? Geef hier ook eventueel gewenste extra onderwerpen aan.

Deelonderwerpen in programmeeronderwijs	Hoeveelheid aandacht voor het onderwerp in de methode			
	veel	niet veel niet weinig	weinig	geen
1. het leren over programmeertalen				
2. het leren over de werking v/e computer				
3. algoritmen				
4. problemen opdelen in deelproblemen				
5. programma-ontwerp				
6. het maken van een programma				
7. het testen van een programma				
8. het leren van een programmeertaal				
9.				
10.				

Bent u het eens met de gekozen nadrukken (nog los van de manier van uitwerking – zie hierna)?

Deelonderwerpen in programmeeronderwijs	Hoeveelheid aandacht voor het onderwerp in de methode			
	te veel	net goed	te weinig	veel te weinig
1. het leren over programmeertalen				
2. het leren over de werking v/e computer				
3. algoritmen				
4. problemen opdelen in deelproblemen				
5. programma-ontwerp				
6. het maken van een programma				
7. het testen van een programma				
8. het leren van een programmeertaal				
9.				
10.				

Wat vindt u van de manier waarop de onderwerpen aan de orde komen?

Deelonderwerpen in programmeeronderwijs	Manier van uitwerking			
	goed	voldoende	matig	zeer slecht
1. het leren over programmeertalen				
2. het leren over de werking v/e computer				
3. algoritmen				
4. problemen opdelen in deelproblemen				
5. programma-ontwerp				
6. het maken van een programma				
7. het testen van een programma				
8. het leren van een programmeertaal				

Geef eventuele toelichting aan ommezijde.

Toelichting/opmerkingen:

Analyseformulier practica

Practicum:…….

a. Didactische aanpak

(1) spiral approach, (2) top-down, (3) bottom-up, (4) aanvullen en/of lezen, of (5) een combinatie van enkele van deze aanpakken (geef aan welke).

b. Sterke/zwakke kanten

Sterk:……...…
……
……

Zwak:……..……
……
……

c. Noodzakelijke veranderingen:

:……..……
……
……

d. Aandacht voor deelonderwerpen

Deelonderwerpen in programmeeronderwijs	Wordt geoefend in het practicum?	
	Ja	Nee
1. het leren over programmeertalen		
2. het leren over de werking v/e computer		
3. algoritmen		
4. problemen opdelen in deelproblemen		
5. programma-ontwerp		
6. het maken van een programma		
7. het testen van een programma		
8. het leren van een programmeertaal		
9.		
10.		

Bijlage 6 Literatuurverwijzingen

[1] *Advies Examenprogramma's havo/vwo Informatica*, Stuurgroep Profiel Tweede Fase, Den Haag, 1995.

[2] *Programmeertalen en -paradigma's*, Jeroen Fokker en Willem van der Vegt, Codi-cursus, 1999

[3] *From the Bottom to the Top?,* Betsy van Dijk en Herman Koppelman (1995), TINFON 4 (1), pp. 31-32. Zie ook bijlage 1.

[4] *Programmeren door completeren,* Jeroen van Merriënboer (1992), TINFON 1 (2), pp. 66-71. Zie ook bijlage 2.

[5] *Visual programming with Java: an alternative approach to introductory programming*, Frank Wester, Marleen Sint, en Peter Kluit (1997). In Proceedings Integrating Technology into Computer Science Education (ITiCSE 97, Uppsala, Sweden, pp.57-58, ACM (`www.acm.org/pubs/citations/proceedings/cse/268819/p57-wester/`)