Die onderrig van *Scratch* met die oog op die aanleer van *Delphi* as objekgeoriënteerde programmeertaal

SC van Zyl
11053836

Verhandeling voorgelê ter nakoming vir die graad

Magister Educationis in Rekenaarwetenskaponderwys aan die

Potchefstroom Kampus van die Noordwes-Universiteit

Studieleier: **Prof E Mentz**

Medestudieleier: **Dr HM Havenga**

Oktober 2014

VERKLARING

Ek, die ondergetekende, verklaar hiermee dat die werk wat in hierdie tesis bevat is, my eie oorspronklike werk is en dat ek dit nie voorheen in die geheel, of gedeeltelik by enige universiteit vir graaddoeleindes ingedien het nie.

SC van Zyl

23 Oktober 2014

DANKBETUIGINGS

SOLI DEO GLORIA!

Hiermee wil ek graag my opregte dankbetuiging uitspreek teenoor die onderstaande persone en instansies

- Prof Elsa Mentz, vir haar inspirasie, toegewydheid en bekwame leiding.
- Dr Marietjie Havenga, vir haar bekwame leiding, motivering en ondersteuning.
- Me Jackie Viljoen, vir taalversorging, en kontrolering van teksverwysings.
- Me Susan van Biljon, vir tegniese versorging.
- Noordwes Departement van Onderwys, vir toestemming om die navorsing te kon doen.
- Onderwysers wat aan die navorsing deelgeneem het en bereid was om onderhoude toe te staan.
- Prof Barry Richter, Louise en Christelle en die administratiewe personeel vir hulle ondersteuning en aanmoediging.
- My kollegas by die Fakulteit Opvoedingswetenskappe, in die besonder die vakgroep Rekenaarwetenskaponderwys, vir hul aanmoediging en ondersteuning.
- My vriende en familie, vir hul aanmoediging en ondersteuning.
- My eggenoot, Egbert en my kind, Marwileen vir hulle begrip en ondersteuning.
- My Hemelse Vader, vir die geleentheid om te kan studeer en elke dag se onverdiende genade.

OPSOMMING

DIE ONDERRIG VAN *SCRATCH* MET DIE OOG OP DIE AANLEER VAN *DELPHI* AS OBJEKGEORIËNTEERDE PROGRAMMEERTAAL

Volgens die Kurrikulum- en Assesseringsbeleidsverklaring moet Inligtingstegnologie-leerders in graad 10 aanvangsonderrig in *Scratch* as programmeertaal ontvang en in graad 11 na *Delphi* of *Java* as programmeertaal oorskakel.

In Noordwesprovinsie word *Delphi* as programmeertaal in graad 11 bestudeer. Die aard van *Scratch* en dié van *Delphi* verskil egter van mekaar. *Scratch* is ’n visuele programmeertaal met ’n speelse aard, waar boublokke met ingeboude programmeringskode inmekaar gepas word om programme te skep, terwyl programmeringskode in *Delphi,* onderhewig aan sintaktiese reëls, ingetik word. Daar word verder van leerders verwag om by *Delphi* ’n objekgeoriënteerde programmeringsbenadering te volg terwyl *Scratch* as ’n objekgebaseerde programmeertaal beskou word.

Die oorgang vanaf *Scratch* na *Delphi* is nog nie tevore in navorsing ondersoek nie. Die doel van hierdie navorsing was dus om te bepaal hoe *Scratch* aan graad 10-leerders onderrig behoort te word om die oorgang na *Delphi* as sintaksisgebaseerde, objekgeoriënteerde programmeertaal te ondersteun. ’n Kwalitatiewe studie is gedoen waartydens onderhoude met agt Inligtingstegnologie-onderwysers in Noordwesprovinsie gevoer is.

Aan die begin van 2013 is die eerste onderhoude gevoer om te bepaal hoe *Scratch* in graad 10 onderrig is. Ná verloop van ses maande is ’n tweede onderhoud met elke deelnemende onderwyser gevoer om te bepaal hoe die oorgang na *Delphi* in graad 11 ervaar is. Die bevindings dui daarop dat deelnemende onderwysers onseker was hoe om *Scratch* te onderrig en dat programmeringsbeginsels en -begrippe nie pertinent onderrig is nie. Onderwysers het op leerders se intuïtiewe begrip van programmeringsbegrippe staatgemaak en geskikte onderrig–leerstrategieë is nie toegepas nie. Met die oorgang na *Delphi* in graad 11 het onderwysers aangedui dat hulle programmering feitlik weer van voor af moes onderrig en dat baie min programmeringsbegrippe na *Delphi* oorgedra is. Onderwysers was onder druk om

programmeringsbegrippe wat in graad 10 reeds vasgelê moes gewees het, tesame met nuwe begrippe wat vir die onderrig van *Delphi* benodig was, te onderrig.

Voorstelle van onderwysers om toekomstige druk te verlig, het betrekking gehad op ’n inkorting in *Scratch*-onderrigtyd en was nie gerig op die onderrig van programmeringsbeginsels en -begrippe in *Scratch* met die oog op die oorgang na *Delphi* nie. Tydens die navorsing is bevind dat die meeste programmeringsbeginsels en -begrippe reeds met *Scratch* onderrig kan word. *Scratch*-onderrig, met die oog op die oorgang na *Delphi,* moet egter noodwendig anders geskied as *Scratch*-onderrig ten einde ’n belangstelling vir programmering te kweek. Na aanleiding van hierdie navorsing word aanbevelings gemaak om *Scratch* te onderrig, sodat die kennis en vaardighede wat graad 10-leerders met *Scratch* opdoen, verband hou en toegepas kan word op *Delphi*. Om onderwysers te ondersteun en hulle onsekerheid van *Scratch*-onderrig die hoof te bied, word spesifieke aanbevelings met betrekking tot die onderrig van *Scratch* gemaak ten opsigte van probleemvoorstelling en algoritme-ontwerp, instandhouding, foutopsporing en fouthantering, veranderlikes en datatipes, herhaling- en besluitnemingstrukture, objekgeoriënteerde begrippe, bewerkingsoperators, karakterhantering en ’n onderrig–leerstrategieë vir *Scratch*.

Sleutelwoorde: *Delphi*, *Scratch*, objekgeoriënteerde programmering, onderrig van programmering, onderrig–leerstrategieë vir programmering, sintaksisgebaseerde programmeertaal, visuele programmeertaal

SUMMARY

THE TEACHING OF *SCRATCH* WITH A VIEW TO LEARNING *DELPHI* AS AN OBJECT-ORIENTED PROGRAMMING LANGUAGE

According to the Curriculum and Assessment Policy Statement, Grade 10 Information Technology learners have to start with programming education by means of the *Scratch* programming language and then change over in Grade 11 to the programming language *Delphi* or *Java*.

In North-West province, *Delphi* is studied as programming language in Grade 11. The characteristics of *Scratch* and *Delphi* differ widely. *Scratch* is a visual programming language, with a playful character, where building blocks with built-in program codes are snapped together to create programs. In *Delphi*, programming code has to be typed in according to syntactic rules. Learners are also expected to follow an object-oriented approach when programming in *Delphi* while *Scratch* can be seen as an object-based programming language.

The transition from *Scratch* to *Delphi* has not been researched before. The purpose of this research was to determine how *Scratch* could be taught to Grade 10 learners, to support the transition to *Delphi* as syntax-based, object-oriented programming language. A qualitative study was done and interviews were held with eight Information Technology teachers in North-West.

The first interviews were held at the beginning of 2013, to determine how *Scratch* was taught in Grade 10. After six months, a second interview was held with each participating teacher, to determine how the transition to *Delphi* was experienced. The findings show that participating teachers were uncertain how to teach *Scratch* and that programming principles and concepts were not specifically taught. Teachers relied on learners' intuitive understanding of programming concepts, and suitable teaching–learning strategies were not applied. With the transition to *Delphi*, teachers indicated that they almost had to start teaching programming all over again and that few programming concepts had been transferred to *Delphi*. Teachers were under pressure to teach programming concepts that should have been established in Grade 10 as well as new concepts that were needed in *Delphi*.

Suggestions that teachers made to relieve future pressure, related to shorter time spent on teaching *Scratch* and were not focused on teaching *Scratch* with a view to transition in *Delphi*. In this research, it was found that most programming principles and concepts can already be taught with *Scratch*. *Scratch* with a view to transition in *Delphi*, will however be taught differently from teaching *Scratch* in order to foster an interest in programming. In this research, recommendations are made for teaching *Scratch*, so that the knowledge and skills that Grade 10 learners obtain are related to and can be applied in *Delphi*. To support teachers and to lessen their uncertainty regarding the teaching of *Scratch*, specific recommendations are made for teaching *Scratch* with regard to problem representation, algorithm design, maintenance, error tracing and -handling, variables and data types, repetition- and decision structures, concepts regarding object orientation, mathematical operators, character handling and a teaching–learning strategy for *Scratch*.

Key words: *Delphi*, *Scratch*, object-oriented programming, syntax-based programming language, teaching–learning strategy for programming, teaching programming, visual programming language

LYS VAN AFKORTINGS

DBE	Departement of Basic Education
DBO	Departement van Basiese Onderwys
DoE	Departement of Education
GK	Grafiese koppelvlak
GOO	Geïntegreerde ontwikkelingsomgewing
IT	Inligtingstegnologie
MIT	Massachusetts Institute of Technology
NKV	Nasionale Kurrikulumverklaring
KABV	Kurrikulum- en Assesserings-beleidsverklaring
OOP	Objekgeoriënteerde programmeertaal
PBL	Probleemgebaseerde leer
RID	Raaksien, insamel en dink
SOOP	Sintaksisgebaseerde objekgeoriënteerde programmeertaal
TVA	Toevoer, verwerking en afvoer
VP	Visuele programmeertaal
VTO	Vinnige toepassinsontwikkeling

INHOUDSOPGAWE

LYS VAN TABELLE

LYS VAN FIGURE

HOOFSTUK 1:

ORIËNTERING

1.1 ALGEMENE PROBLEEMSTELLING

In Suid-Afrika kan leerders vanaf graad 10 Inligtingstegnologie (IT) as vak kies. Die vak behels die studie van tegnologieë wat gebruik word tydens die vaslegging van data, die verwerking van data tot inligting, sowel as die bestuur, aanbieding en verspreiding van data (DBE, 2011:8) en sluit rekenaarprogrammering en die aanleer van ’n programmeertaal in. In die verlede is leerders vanaf graad 10 tot graad 12 in dieselfde programmeertaal onderrig. Vanaf 2005 tot 2011 is leerders in graad 10 tot graad 12 in Noordwesprovinsie in *Delphi* as programmeertaal onderrig. Volgens die Kurrikulum- en Assesserings-beleidsverklaring (KABV) (DBE, 2011) vir IT, moet graad 10 IT-leerders vanaf 2012 programmering deur middel van ’n visuele programmeertaal (VP), *Scratch,* aanleer. In graad 11 moet hierdie leerders oorskakel na ’n sintaksisgebaseerde objekgeoriënteerde programmeertaal (SOOP), naamlik *Delphi* of *Java* en voortbou op programmeringsbegrippe wat met *Scratch* aangeleer is.

Uit die voorafgaande kan die volgende probleemstelling vir hierdie studie geformuleer word:

Op watter wyse behoort programmering in *Scratch* aan graad 10 IT-leerders onderrig te word om die oorgang na die aanleer van *Delphi* as SOOP in graad 11 te ondersteun?

Vir die doel van hierdie studie is daar besluit om slegs op die oorgang van *Scratch* na *Delphi* as programmeertaal te fokus aangesien die verskille tussen die onderrig van *Delphi* en *Java* waarskynlik verskillende benaderings tot die onderrig van *Scratch* regverdig.

1.1.1 Agtergrond tot probleemstelling

Verskeie programmeertale en programmeringomgewings is deur die jare gebruik om leerders in die skool te leer programmeer (Kelleher & Pausch, 2005:85–86; Malan & Leitner, 2007:224). In Suid-Afrika is leerders in die laat 1980's eers deur middel van *Logo* onderrig, 'n programmeertaal wat visuele afvoer gelewer het, voordat hulle na *Basic*, 'n sintaksisgebaseerde programmeertaal, in die vak Rekenaarstudie oorgeskakel het. In die 1990's is die programmeertaal verander na *Pascal*, 'n sintaksisgebaseerde programmeertaal wat spesifiek ontwerp is om gestruktureerde programmering te onderrig (Kelleher & Pausch, 2005:99). In 2005, met die bekendstelling van die nasionale kurrikulum, is die programmeertaal vir die vak IT weer verander na 'n sintaksisgebaseerde objekgeoriënteerde programmeertaal (SOOP) (DoE, 2003:10). Aangesien daar nie konsensus oor die keuse van 'n SOOP tussen die provinsies bereik kon word nie, het die Wes-Kaap, Mpumalanga en KwaZulu-Natal op *Java*, as programmeertaal besluit, en Noordwes, Gauteng, Limpopo, Oos-Kaap, Noord-Kaap en Vrystaat besluit om programmering deur middel van *Delphi* te onderrig vir die vak IT vanaf graad 10 tot 12. In 2013 het Mpumalanga ook oorgeskakel na Delphi.

Tabel 1.1 vergelyk die konseptuele progressie van die Nasionale Kurrikulumverklaring van 2005 (NKV) teenoor die KABV van 2011. In die NKV het leerders *Delphi*-onderrig vanaf graad 10 ontvang en in graad 11 is voortgebou op die begrippe aangeleer in graad 10. In die KABV word programmering in graad 10 deur middel van *Scratch* as programmeertaal onderrig, en in graad 11 deur middel van *Delphi* of *Java.* Programmeringsbegrippe moet dus in graad 10 sodanig met *Scratch* onderrig word, sodat in graad 11 daarop voortgebou kan word, en leerders die oorgang na gevorderde programmering met *Delphi* kan maak.

Tabel 1.1: Vergelyking van progressie tussen NKV en KABV

NKV (2005)		
Graad 10	**Graad 11**	**Graad 12**
Inleiding tot *Delphi* →	**Gevorderde** programmering met *Delphi* →	Gevorderde programmering met *Delphi*
KABV (2012)		
Graad 10	**Graad 11**	**Graad 12**
Inleiding tot programmering d.m.v. *Scratch*	(Nie inleiding tot *Delphi* nie)	
↘	**Gevorderde** programmering met *Delphi* →	Gevorderde programmering met *Delphi*

Die rede vir die voortdurende verandering in programmeertaal en/of programmeeromgewing op skoolvlak kan aan verskeie faktore toegeskryf word, naamlik die veranderende tegnologie, kurrikulumontwikkelaars wat die arbeidsmark tevrede wil stel, eise van leerders wat meer relevante programmeertale wil bestudeer, en die verandering vanaf 'n prosedurele na 'n objekgeoriënteerde paradigma (Sajaniemi & Kuittinen, 2008:75). Alhoewel die keuse van programmeertale beïnvloed word deur die neiging in die arbeidsmark, moet in gedagte gehou word dat programmeertale primêr vir professionele gebruik ontwikkel is en nie om die aanleer van programmering te ondersteun nie (Gomes & Mendes, 2007:2).

Uiteenlopende benaderings kenmerk die onderrig van 'n VP soos *Scratch* teenoor 'n SOOP soos *Delphi*. Leerders kan in *Delphi* 'n prosedurele benadering of 'n objekgeoriënteerde benadering volg (Sebesta, 2012:97) en moet 'n nuwe taal, met semantiek en sintaksis, aanleer. Die programmeringskode moet korrek ingesleutel word, volgens die reëls van die taal (Malan & Leitner, 2007:223), alvorens die program kan uitvoer. Dit impliseer dat leerders eers foute moet identifiseer en regstel, voordat programme kan uitvoer. Daarbenewens is die volgorde van die instruksies en logiese beredenering van die oplossing ook belangrike vaardighede

waarmee leerders moet rekening hou. *Scratch* is egter nie ’n sintaksisgebaseerde taal nie, maar gebruik ’n visuele benadering (Lee, 2011:27; Malan & Leitner, 2007:223). Die *Scratch*-omgewing bestaan uit karakters, wat *Sprites* genoem word en word geprogrammeer om op ’n agtergrond, wat ’n *Stage* genoem word, sekere take uit te voer. Programmering in *Scratch* behels die insleep van bepaalde blokke gegewe kode wat soos stukke van ’n legkaart inmekaar pas en die kode word letterlik deur die leerder ‘gebou’ (Lee, 2011:27). *Scratch* onderskryf sommige beginsels van objekgeoriënteerde programmering, aangesien *Sprites* gedrag en eienskappe het. *Scratch* is egter nie ’n ware objekgeoriënteerde programmeertaal nie, omdat dit nie klasse bevat nie en oorerflikheid nie geprogrammeer kan word nie (Maloney *et al*., 2010:10). Gevolglik word ’n prosedurele benadering in *Scratch*-programmering gevolg. Soos hierbo aangetoon, bestaan daar verskille tussen die twee programmeertale *Scratch* en *Delphi*, wat uiteraard verskillende onderrigbenaderings regverdig. Tabel 1.2 gee ’n uiteensetting van programsegmente in beide *Scratch* en *Delphi* om aan te toon hoe die sintaksis van die twee tale van mekaar verskil, met betrekking tot dieselfde voorbeeld waar ’n herhalingstruktuur gebruik word.

Tabel 1.2: Vergelyking van *Scratch* en *Delphi*

Scratch	***Delphi***

	procedure TForm1.btnCalcClick(Sender: TObject);
	var
	K : Integer;
	begin
	For K := 1 to 10 do
	Begin
	
	End;
	end;

Blokke kode word deur middel van ’n muis ingesleep en pas soos ’n legkaart in mekaar	Instruksies moet in *Delphi* ingetik word. Elke leesteken moet op ’n spesifieke plek gebruik word.

Dit is baie moeilik om programmering aan te leer (Guzdial, 2004:127) en dit is ’n uitdaging om programmering te onderrig (Gomes & Mendes, 2007:1). Die motivering met die implementering van *Scratch* was juis om leerders in graad 10 op ’n informele

en prettige wyse aan die programmeringsomgewing, programmeerinstruksies en programlogika bekend te stel. In graad 11 moet leerders kan bou op die aangeleerde programmeringsbeginsels (DBE, 2011:12). Die gevaar bestaan egter dat vaardighede en begrippe wat noodsaaklik is vir programmering in *Delphi* nie aangeleer word wanneer *Scratch* as eerste programmeertaal gebruik word nie. In die laat 1970's het Clancy (2004:92) gepoog om sy leerlinge eers in 'n eenvoudige programmeertaal, *Karel die Robot*, te onderrig, voordat hulle na 'n formele hoëvlaktaal, *Pascal*, oorgeskakel het. Alhoewel leerders honderde reëls se programmeringskode in *Karel die Robot* kon skryf, kon hulle nie die begrip van veranderlikes en parameter-oordrag in *Pascal* verstaan nie, aangesien *Karel die Robot* nie veranderlikes gebruik het nie. Clancy (2004:92) waarsku dat dit nie as vanselfsprekend aanvaar kan word dat leerders programmeringsbegrippe verstaan nie. Vygotski (1963:24) het bevind dat, indien leer op een domein plaasgevind het, dit nie noodwendig beteken dat vaardighede op ander soortgelyke domeine ook toegepas kan word nie. In navorsing wat hy gedoen het, kon volwassenes wat geleer het om lengte van kort strepe te skat, nie hulle vaardighede toepas om lengte van lang strepe ook te skat nie (Vygotski, 1963:24). Vaardighede soos redenasie en observasie word deur Vygotski (1963:25) egter as basiese vaardighede gesien, wat wel verbeter en versterk kan word indien dit van een situasie of vak na 'n ander oorgedra word.

Carver (1986:12) het reeds in 1986 bewys dat dit moontlik is om vaardighede van een programmeertaal na 'n ander oor te dra, mits die onderrig spesifiek op die oordrag van vaardighede gerig is. Die onmiddellike voordeel hiervan is 'n besparing in onderrigtyd, aangesien begrippe nie weer in die nuwe programmeringsomgewing onderrig hoef te word nie (Carver, 1986:67). Hieruit kan afgelei word dat die onderrig in *Scratch* spesifiek op die oordrag van vaardighede na *Delphi* gerig moet word om die werkslading in die korter tyd wat vir *Delphi* toegeken word (nou net in graad 11 en graad 12, sien tabel 1.1), af te handel. Vygotski (1963:30) het aangetoon dat onderrig wat slegs deur middel van die visuele gedoen word, die vorming van abstrakte denke belemmer. Aangesien *Scratch*-leerders in 'n visuele omgewing leer programmeer, is dit noodsaaklik om verdere ondersoek in te stel hoe om in hierdie visuele omgewing te onderrig en leerders se abstrakte denke te ontwikkel. Abstrakte programmeringsbegrippe is 'n hoë kognitiewe vaardigheid wat reeds van die begin af

deur leerders toegepas moet word om konkrete probleme op te los (Gomes & Mendes, 2007:1). Armoni *et al*. (2006:283) beskou ook abstraksie as ’n denkvaardigheid wat noodsaaklik is vir programmering en stel dit duidelik dat dit vir die oplossing van komplekse probleme benodig word.

Die finale woord behoort aan Kelleher en Pausch (2005:131):

"[P]erhaps it is time to begin studying the intermediate programmer, someone who has been introduced to programming through a system designed for beginners and wants to apply that experience to learning a general language. What are the hardest aspects of that transition and how are those aspects affected by the teaching system? What are the trade-offs between presenting issues of syntax and program expression earlier or later in the process?"

Uit bostaande argumente kan afgelei word dat daar nie sonder meer aanvaar kan word dat oordrag van programmeringsbeginsels en –begrippe vanaf een programmeertaal na ’n ander sal plaasvind nie. Die vraag kan dus gevra word hoe onderwysers *Scratch* as VP moet onderrig met die oog op die toekomstige aanleer van *Delphi* in graad 11.

1.2 OORSIG VAN RELEVANTE LITERATUUR

Die aanleer van programmering kan in dieselfde kategorie gesien word as die aanleer van ’n geskrewe taal of wiskunde, aangesien ’n taal met ’n bepaalde sintaksis en semantiek aangeleer moet word (Gomes & Mendes, 2007:1).

Die aanleer van die sintaksis en semantiek van ’n programmeertaal is ’n uitdaging tydens die onderrig van programmering (Gomes & Mendes, 2007:2; Malan & Leitner, 2007:223). Taalkundig verwys sintaksis na die manier waarop ons woorde bymekaarvoeg om sinne te vorm (Sternberg & Sternberg, 2012:367), terwyl sintaksis in programmeertale dui op die vorm van uitdrukkings en programeenhede (Sebesta, 2012:116). Taalkundig word semantiek beskou as die studie van die betekenis van taaluitinge (Sternberg & Sternberg, 2012:368), en in programmeertale dui dit op die betekenis van instruksies en programeenhede (Sebesta, 2012:116). Omdat die sintaksis in programmeertale so omvattend is, konsentreer onderwysers eerder op die onderrig van die sintaksis as op probleemoplossing, en in die proses word die

basiese begrippe van programmering nie aangeleer nie (Gomes & Mendes, 2007:1–2). Dit is juis die rede wat aangevoer word vir die implementering van *Scratch* vir graad 10 IT-leerders, aangesien die fokus nie hier op sintaksis is nie en daar primêr op die begrippe van programmering, programmeringsvaardighede, algoritme-ontwerp en probleemoplossing gefokus kan word (DBE, 2011:12). In ’n SOOP moet programstrukture, soos bv. die FOR-lus en IF-stelling aan leerders verduidelik word, en leerders moet die komplekse sintaksis daarvan memoriseer voordat hulle dit in programme kan gebruik (Gomes & Mendes, 2007:2). Leerders moet dus op probleemoplossing, algoritme-ontwerp en sintaksis konsentreer wanneer hulle programme skryf (Gomes & Mendes, 2007:1). Beginnerprogrammeerders vind die kombinasie van probleemoplossing en sintaksis baie verwarrend en dit is vir hulle moeilik om dit wat hulle met die program wil doen, in sintaksis korrekte stellings wat die rekenaar kan verstaan te vertaal (Kelleher & Pausch, 2005). Verder benodig leerders met ’n SOOP ook goeie sleutelbordvaardighede (Lee, 2011:27) om die kode akkuraat in te sleutel, aangesien elke kommapunt belangrik is. Die gevolg is “students become masters of syntax before solvers of problems” (Malan & Leitner, 2007:223). Uit die literatuur kan dus afgelei word dat daar voortdurend ’n stryd is tussen die aanleer van die sintaksis aan die een kant, en die vorming van hoë kognitiewe denke om ’n program te skryf en probleme op te los, aan die ander kant.

Programmering is ’n moeilike vaardigheid om aan te leer en ’n moeilike vaardigheid om te onderrig (Traynor & Gibson, 2004:1), ongeag die ouderdom van die persoon wat dit probeer aanleer (Gomes & Mendes, 2007:1; Guzdial, 2004:128; Kelleher & Pausch, 2005:83). Alhoewel Traynor en Gibson (2004:1) beweer dat navorsing oor die onderrig van programmering van kardinale belang is, voer Guzdial (2004:129) aan dat navorsers die vrae hoe leerders leer programmeer en wat die voorvereistes vir suksesvolle aanleer van programmering is, geringskat. Volgens Gomes en Mendes (2007:1) is verskeie oplossings reeds voorgestel om die aanleer van programmering te ondersteun, maar daar is nie een benadering wat algemeen toegepas kan word nie.

Om ’n programmeringsprobleem op te los, moet die probleem geformuleer word, geanaliseer word, ’n oplossing ontwerp word en kode vir die oplossing geskryf word (Grant, 2003:96). Leerders sukkel in die algemeen om probleme op te los en algoritmes te ontwikkel. Hulle verstaan nie probleme nie, kan nie vorige kennis na

nuwe probleme oordra nie en reflekteer nie oor oplossings nie (Gomes & Mendes, 2007:2). Hoe beter die probleme wat leerders tydens die aanleer van programmering ervaar verstaan word, hoe beter kan leerders onderrig en ondersteun word (Havenga, 2008:80). Havenga (2008:72) meld dat beginnerprogrammeerders tydens programmering foute maak as gevolg van die kompleksiteit van programmering, en klassifiseer die moontlike oorsake vir programmeerfoute onder die kategorieë kognitief, metakognitief en kwessies rakende probleemoplossing. Baldwin en Kuljis (2000:285) beweer dat onderwysers bewus moet wees van kognitiewe vaardighede en hoe leer plaasvind wanneer programmering onderrig word en nie net op die programmeringsvaardighede wat ontwikkel moet word moet fokus nie. Havenga (2008:iii) sluit hierby aan deur te noem dat kognitiewe vaardighede sowel as programmeringsvaardighede pertinent onderrig moet word en dat daar nie aangeneem kan word dat hoëvlakkennis, vaardighede en strategieë vanself by leerders sal ontwikkel nie. Kognitiewe vaardighede soos redenasievermoë, analitiese denke, logiese denke en probleemoplossingsvaardighede is deur Ismail *et al*. (2010:128) as noodsaaklike vaardighede vir rekenaarprogrammering, geïdentifiseer.

Al is die kennis en vaardighede wat leerders vir programmering benodig 'n onderwerp van bespreking onder navorsers (Baldwin & Kuljis, 2000:285; Grant, 2003:97; Guzdial, 2004:131), word daar volgens Baldwin en Kuljis (2000:286) algemeen aanvaar dat leerders vir die aanleer van programmering drie soorte kennis benodig, naamlik sintaktiese kennis, konseptuele kennis en strategiese kennis. Wanneer leerders sintaktiese kennis het, kan hulle wel 'n program skryf, maar het nog nie die kennis om programme te ontwerp wat probleme kan oplos nie. Konseptuele kennis behels die begrip van beginsels aangaande die semantiek van programmering. Strategiese kennis is die mees komplekse kennis wat leerders vir programmering benodig, wat beteken dat leerders wat hierdie kennis besit, ingewikkelde programmeringsprobleme kan oplos, aangesien hulle dan logiese oplossings vir komplekse programme kan beplan (Baldwin & Kuljis, 2000:286).

Die *Scratch*-programmeertaal kan gebruik word vir die ontwikkeling van strategiese kennis, aangesien dit juis op die oplossing van probleme fokus en nie op die begrip van sintaksis nie (Malan & Leitner, 2007). Leerders word aan al die begrippe van programmering blootgestel en kan met *Scratch* eksperimenteer, sonder om sintaksisfoute te maak (Kelleher & Pausch, 2005:95). Wanneer 'n *Scratch*-program

uitgevoer word, kan die logiese volgorde van programinstruksies deur die leerder waargeneem word (Maloney *et al*., 2010:3), wat hoërorde-denkvaardighede en konkrete denke kan stimuleer (Kelleher & Pausch, 2005:103). Die leerders kan sodoende oor hulle oplossings dink en veranderings maak wanneer programme nie die korrekte afvoer lewer nie. Foutiewe programuitvoering word onmiddellik waargeneem en leerders kan veranderings aan die kode maak en die gevolge daarvan sien (Maloney *et al*., 2010:3).

Die aanvanklike doel met *Scratch* was om ’n maklike programmeringsplatform te ontwerp, met die fokus op animasie en genot, sodat jong leerders van agt jaar oud af aan programmering blootgestel kan word (Utting *et al*., 2010:2). Die vraag of *Scratch* egter ’n geskikte taal vir die onderrig van programmering is, was buite die veld van die studie. Met ’n VP, soos *Scratch*, kan multimedia-toepassings maklik deur leerders geskep word (Lee, 2011:28) en leerders kan animasie, speletjies en interaktiewe kuns deur middel van *Scratch* ontwikkel (Malan & Leitner, 2007:223). Utting *et al.* (2010:3) beweer *Scratch* kan daartoe bydra om meer leerders in programmering te interesseer. Hierdie navorsing was egter beperk tot die onderrig van *Scratch* en nie die uitwerking van *Scratch* op leerders se gesindheid teenoor programmering nie.

Kritiek wat soms teen *Scratch* geopper word, is dat werkende programme geskep kan word deur blokke kode met behulp van probeer en tref inmekaar te pas en leerders nie noodwendig hierdeur leer om hulle programme te beplan nie (Utting *et al*., 2010:4). Sajaniemi en Kuittinen (2008:80) beweer egter dat sekere aspekte van programmering, soos sintaksis, nie opsygeskuif kan word wanneer programmering aanvanklik aangeleer word nie en stel dit onomwonde dat “[p]rogramming should not be taught as a copy-and-paste art that only incidentally results in a correctly functioning program”. Leerders moet verder spesifiek geleer word hoe om te bou op wat hulle deur middel van probeer en tref in *Scratch* ontdek het (Utting *et al*., 2010:4). Onderrig deur middel van *Scratch* moet deeglik beplan word en op probleemoplossingstrategieë gerig word, sodat leerders kan verstaan watter logiese foute hulle maak en nie net op probeer en tref steun om programfoute op te los nie (Utting *et al*., 2010:5). Indien leerders op probeer en tref metodes steun, sal strategiese kennis nie verkry word nie. Om hierdie strategiese kennis te verkry sodat oplossings vir probleme geformuleer kan word, beweer Baldwin en Kuljis (2000:288) dat ’n beginnerprogrammeerder op ’n visuele manier moet leer programmeer, maar

terselfdertyd ook ondersteun moet word om die onderliggende begrippe soos veranderlikes, bewerkings, beheervloei en subprogramme te verstaan. Traynor en Gibson (2004:2) beweer dat leerders dikwels nie kan programmeer nie, omdat bepaalde onderliggende programmeringsbeginsels ontbreek. Leerders sien dikwels probleemoplossing en kodering (die skryf van programmeringskode) as een en dieselfde, en sien nie die verband en onderlinge afhanklikheid tussen probleemoplossing en kodering raak nie (Traynor & Gibson, 2004:2).

Wanneer programmering in ’n SOOP onderrig word, word ’n ander benadering gebruik as met die onderrig van ’n VP. In ’n SOOP word die sintaksis van die taal eerste aangeleer; daarna word daar geleidelik na die semantiek beweeg met die hoop dat ’n leerder later ware programmeringsvaardighede sal aanleer (Jenkins, 2002:55). Die voorstelling van die oplossing vir ’n probleem met ’n algoritme word volgens Jenkins (2002:55) as die moeilikste deel van programmering beskou. Tydens die onderrig van ’n SOOP word die onderrig van die sintaksis van die programmeertaal egter ten koste van algoritme-ontwerp oorbeklemtoon. Die skryf van algoritmes word dikwels na ’n ander afdeling van die leerinhoud geskuif, wat nie met programmering in verband gebring word nie Verder is onderrig wat op die aanleer van sintaksis fokus, gewoonlik vervelig en dit verhoog nie die genotfaktor van programmering nie (Jenkins, 2002:56).

Dit blyk dus uit bostaande dat daar ’n verskil tussen die onderrigbenadering van ’n VP en dié van ’n SOOP is en dat daar verskillende strominge ten gunste van elk bestaan. Die oogmerk van hierdie studie was egter om ’n middeweg te vind sodat aanvangsprogrammeringonderrig in ’n VP die latere onderrig van ’n SOOP kan bevoordeel.

1.3 DOEL MET DIE NAVORSING

Die doel met die navorsing was om te bepaal hoe *Scratch* as ’n visuele programmeertaal (VP) aan graad 10-leerders onderrig behoort te word om die oorgang na *Delphi* as sintaksisgebaseerde objekgeoriënteerde taal (SOOP) te vergemaklik.

Ten einde die doel te bereik is die onderstaande subdoelwitte vir hierdie studie gestel:

1. om te bepaal wat die aard van *Scratch* as programmeertaal is en hoe dit onderrig behoort te word;

2. om te bepaal wat die aard van *Delphi* as programmeertaal is;

3. om te bepaal watter algemene beginsels van programmering reeds tydens die onderrig van *Scratch* vasgelê kan word;

4. om te bepaal hoe die algemene beginsels van programmering tydens die onderrig van *Scratch* onderrig behoort te word met die oog op die latere bemeestering van *Delphi;*

5. om te bepaal hoe *Scratch* tans in Noordwesprovinsie waar *Delphi* in graad 11 geïmplementeer word, onderrig word;

6. om te bepaal hoe onderwysers die oorgang vanaf die onderrig van *Scratch* na die onderrig van *Delphi* ervaar het; en

7. om te bepaal watter riglyne aan onderwysers gegee kan word waarvolgens hulle leerders tydens die onderrig van *Scratch* kan ondersteun om die oorgang na *Delphi* as programmeertaal vir die leerders makliker te maak.

Die onderstaande navorsingsvrae is in die studie ondersoek:

1. Wat is die aard van *Scratch* as programmeertaal en hoe behoort dit onderrig te word?

2. Wat is die aard van *Delphi* as programmeertaal?

3. Watter algemene beginsels van programmering kan reeds tydens die onderrig van *Scratch* vasgelê word?

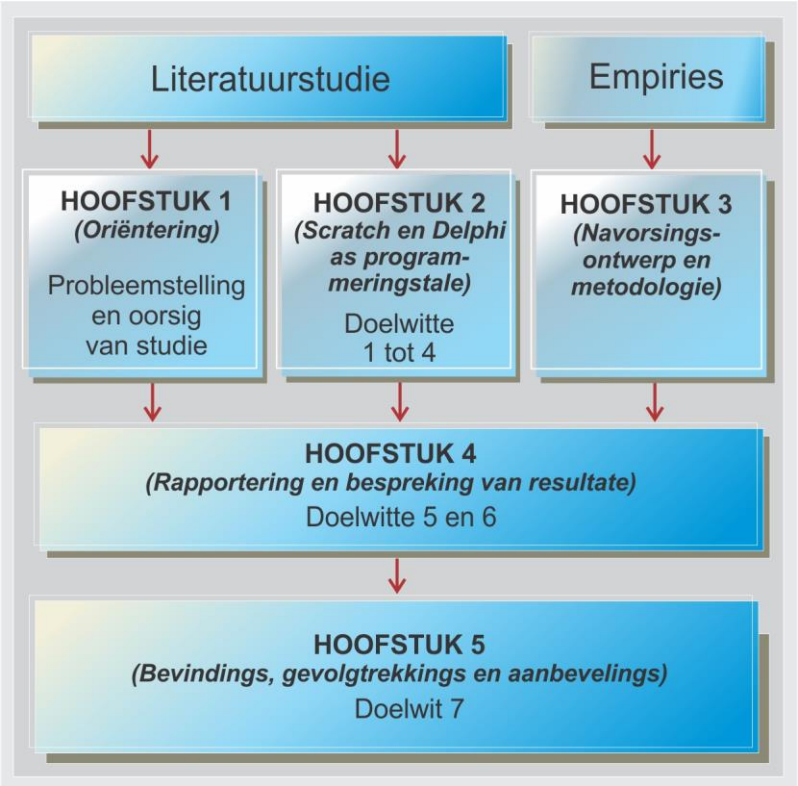
4. Hoe behoort die algemene beginsels van programmering tydens *Scratch* onderrig te word met die oog op die latere bemeestering van *Delphi?*

5. Hoe word *Scratch* tans in Noordwesprovinsie waar *Delphi* in graad 11 geïmplementeer word, onderrig?

6. Hoe het onderwysers die oorgang vanaf die onderrig van *Scratch* na die onderrig van *Delphi* ervaar?

7. Watter riglyne kan aan onderwysers gegee word om leerders tydens die onderrig van *Scratch* te ondersteun om die oorgang na *Delphi* as programmeertaal vir die leerders makliker te maak?

Figuur 1.1 gee 'n diagrammatiese voorstelling van die hoofstukindeling van die studie en toon aan hoe die bogenoemde subdoelwitte aangespreek gaan word.



Figuur 1.1: Diagrammatiese voorstelling van hoofstukke

1.4 NAVORSINGSONTWERP

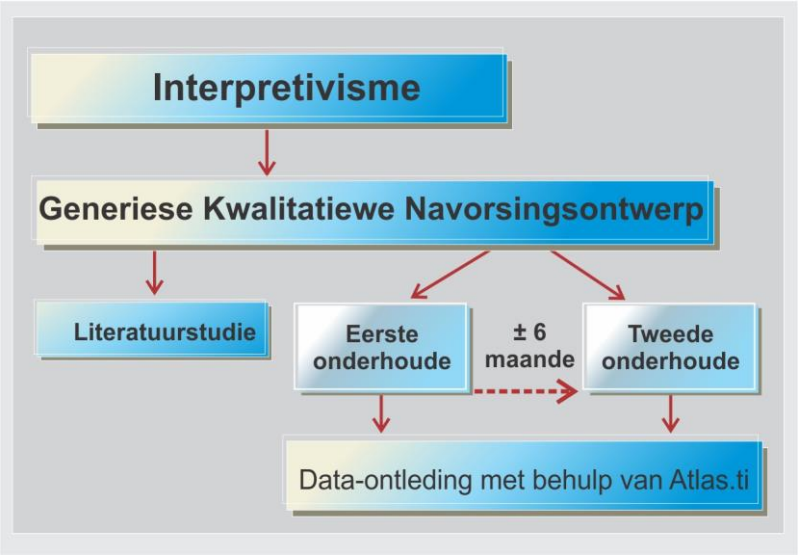
In hierdie afdeling word die navorsingsontwerp van die studie bespreek.

1.4.1 Paradigma en metodologie

Die studie is vanuit 'n interpretivistiese paradigma onderneem. Die hoofdoel met interpretivisme is om dit wat bestudeer word, te verstaan (Olivier, 2009:112). Voordat enige aanbevelings oor die onderrig van *Scratch* gemaak kon word, wou die navorser eers verstaan hoe programmering onderrig moet word en wat onderwysers se ervarings van hulle *Scratch*-onderrig en die oorgang na *Delphi* was. Die hoofdoel met die navorsingsvrae in die studie was dus om te verstaan hoe *Scratch* onderrig

behoort te word, wat onderwysers se refleksie oor hulle *Scratch*-onderrig was en hoe hulle die oorgang na *Delphi* ervaar het. ’n Kwalitatiewe ondersoekende studie is uitgevoer, waar die navorser onderhoude met deelnemers gevoer het om sodoende ’n begrip van dit wat gesê is, op te bou (Creswell, 2008:26). Die navorser het gevolglik probeer om so na moontlik aan deelnemers se situasie te wees, om die wêreld vanuit deelnemers se perspektiewe te beskou (Babbie & Mouton, 2001:33).

Figuur 1.2 toon ’n diagrammatiese uiteensetting van die navorsingsmetode en data-insameling- en data-ontledingmetodes van die navorsing.



Figuur 1.2: Uiteensetting van navorsingsontwerp

1.4.2 Deelnemers

Die teikenpopulasie was al die IT-onderwysers in Noordwes (n=24). ’n Homogene, ewekansige steekproefmetode (Creswell, 2008:214) is gebruik om onderwysers te selekteer (n=10). Homogene steekproefneming word gebruik indien deelnemers ’n sekere gemeenskaplike eienskap moet besit (Creswell, 2008:216), wat in hierdie studie onderwysers was wat in 2012 graad 10 IT-leerders en in 2013 graad 11 IT-leerders onderrig het. Uit hierdie groep onderwysers is 10 onderwysers ewekansig geselekteer om aan die navorsing deel te neem.

1.4.3 Data-insamelingstegnieke

Twee stelle onderhoude is deur die navorser met deelnemers gevoer. Eerstens is semi-gestruktureerde onderhoude oor die onderrig van *Scratch* met deelnemers gevoer. Semi-gestruktureerde onderhoude word gebruik om data te bevestig wat uit ander bronne bekom is deurdat deelnemers voorafopgestelde vrae beantwoord en die navorser deelnemers verder uitvra indien hulle antwoorde nie duidelik is nie (Nieuwenhuis, 2007:87). Op hierdie stadium was sekere riglyne oor programmering reeds uit die literatuur geïdentifiseer en in die onderhoude is na hierdie riglyne verwys om te bepaal hoe deelnemers dit in hulle *Scratch*-onderrig hanteer het. Die onderhoude is aan die begin van 2013 gevoer, voordat onderwysers met *Delphi*-onderrig gevorder het, sodat hulle onderrig van *Delphi* nie hulle siening van hulle *Scratch*-onderrig kon beïnvloed nie. Tydens die onderhoudsgeleentheid is ingeligte toestemmingsvorms voltooi (sien 1.5) en die doel met die navorsing is aan deelnemers verduidelik.

Ná verloop van ongeveer ses maande se *Delphi*-onderrig is ŉ tweede onderhoud met elk van die deelnemers gevoer. In hierdie onderhoude is vrae (Creswell, 2008:213) aan deelnemers gestel om vas te stel hoe hulle die oorgang na *Delphi* ervaar het (Babbie & Mouton, 2001:33). Die doel met die onderhoude was om die onderwysers in staat te stel om hulle ervarings ten opsigte van die oorgang vanaf *Scratch* na *Delphi* met die navorser te deel en sodoende ŉ ryk verskeidenheid data te verkry wat die navorser kon help om onderrigriglyne saam te stel wat die oorgang vanaf *Scratch* na *Delphi* verder kan ondersteun.

1.4.4 Data-ontleding

Alle onderhoude is elektronies opgeneem en getranskribeer (Creswell, 2008:246), sodat dit met die program Atlas.ti gekodeer kon word, om temas wat uit deelnemers se opinies na vore kom, te bepaal. Onderhoude is gevoer totdat data-saturasie voorgekom het en gevolglik is minder as die aangeduide tien onderwysers (sien 1.4.2) in die finale groep ingesluit. Data-saturasie is verkry toe nuwe temas nie meer uit onderhoude identifiseer is nie (Creswell, 2008:443).

Uit die ontleding en interpretering van die temas te same met die literatuurstudie is bevindings, gevolgtrekkings en aanbevelings vir die onderrig van *Scratch* met die oog op die latere onderrig van *Delphi* as programmeertaal, gemaak.

1.4.5 Betroubaarheid en geldigheid

Tydens kwalitatiewe navorsing is die navorser die instrument wat die data 'meet', en interaksie tussen die navorser en die deelnemer is onvermydelik. Gevolglik word objektiwiteit dikwels in dié soort navorsing bevraagteken (Olivier, 2009:112). Die navorser het dus gepoog om die navorsing so objektief moontlik te hou deur nie die onderhoude met sy/haar eie voorkeure te besoedel en deelnemers in die rede te val nie. Vrae is so breed moontlik gestel (Creswell, 2008:129) en deelnemers is nie gelei om bepaalde antwoorde te gee nie.

Volgens Creswell (2009:190) is die akkuraatheid van kwalitatiewe bevindings van uiterste belang. Verskeie maniere word aangedui om die geldigheid en betroubaarheid van navorsing te versterk, soos onder andere triangulasie, die nagaan van temas en die finale verslag deur deelnemers, en die gebruik van 'n eksterne ouditeur om die projek na te gaan (Creswell, 2009:191-192). Morse *et al*. (2002:16) beweer dat dit te laat is indien die navorser resultate ekstern wil verifieer, maar dat betroubaarheid en geldigheid bereik kan word indien die navorser deurgaans alle handelinge verifieer en gedurig kreatief en met insig en openheid daarop reageer. In die navorsing is Morse *et al*. (2002:16) se benadering gevolg en die navorser het gepoog om deurgaans alle handelinge, soos die nagaan van transkripsies, en toekenning van temas, na te gaan en te verfyn.

1.5 ETIESE ASPEKTE

Die navorsing is onder die vleuel van die SANPAD-projek gedoen en toestemming is by die Departement van Onderwys daarvoor verkry (sien addendum A). Toestemming is ook by die NWU se etiese komitee verkry, sowel as by die hoof van elke deelnemende skool (sien addendums B en C).

Deelnemers het 'n toestemmingsvorm onderteken (sien addendum C) om aan die navorsing deel te neem. Hulle is ook ingelig dat deelname vertroulik, anoniem en vrywillig is. Anonimiteit is verseker deurdat die navorser self die onderhoude gevoer

het, transkripsies van onderhoude op veilige plekke bewaar word, die data deur die navorser self ontleed is en deelnemers se name nêrens genoem word nie. Deelnemers het die reg gehad om hulle, sonder enige benadeling, te eniger tyd aan die navorsing te onttrek.

Geen sensitiewe of vertroulike inligting is bekend gemaak nie. Die deelnemers moes wel hulle kwalifikasies en aantal jare ervaring in IT-onderrig bekend maak. Daar word beplan om aan die deelnemers terugvoer te gee oor die uitkoms van die studie, aangesien dit vir hulle onderrig verrykend kan wees.

In die onderhoude is daarop gelet om nie die navorser se eie mening oor die onderrig van *Scratch* en die oorgang na *Delphi* bekend te maak nie (Creswell, 2008:239), sodat die deelnemers se menings nie daardeur beïnvloed sou word nie.

1.6 BYDRAE VAN DIE STUDIE

Scratch is vir die eerste keer in 2012 aan IT-leerders in Suid-Afrika onderrig. Daar is geen loodsprojekte deur die Departement van Onderwys onderneem om behoorlike strategieë vir die onderrig van *Scratch* te bepaal nie. Hierdie studie sal dus ’n bydrae tot die onderrig van IT lewer deur voorstelle vir die onderrig van *Scratch* te maak met die oog op die onderrig van *Delphi* in graad 11, sodat bestaande kennis en vaardighede oorgedra word en leerders nie weer van nuuts af onderrig hoef te word insake elementêre programmeringsbegrippe nie. Die studie dra by tot die SANPAD-navorsingsprojek oor bemagtiging van IT-onderwysers binne die Teaching Learning Praxis-fokusarea as deel van die projek Self-Directed Learning en kan waardevolle aanbevelings vir die verbetering van IT as skoolvak maak.

Aangesien daar in die aanvanklike literatuurstudie genoem word dat die onderrig van programmering moeilik is (Gomes & Mendes, 2007:1; Guzdial, 2004:128; Kelleher & Pausch, 2005:83), lewer hierdie navorsing ’n bydrae in die sin dat riglyne gegee word vir die effektiewe onderrig van *Scratch* wat sodoende die aanleer van programmering kan vergemaklik.

Hoewel hierdie studie slegs op die oorgang vanaf *Scratch* na *Delphi* gefokus het, behoort die aanbevelings wat uit hierdie studie voortspruit ’n waardevolle bydrae tot die oorgang vanaf *Scratch* na *Java* te lewer.

1.7 HOOFSTUKINDELING

Hoofstuk 1: Oriëntering

Hoofstuk 2: *Scratch* en *Delphi* as programmeertale tydens die onderrig van programmering aan IT-leerders

Hoofstuk 3: Navorsingsontwerp en metodologie

Hoofstuk 4: Rapportering en bespreking van resultate

Hoofstuk 5: Bevindings, gevolgtrekkings en aanbevelings

1.8 TYDRAAMWERK

Februarie 2013: Eerste onderhoude

Julie 2013: Tweede onderhoude

November 2013: Data-analise en resultate

April 2014: Rapportering en bespreking van resultate

Junie 2014: Riglyne vir die onderrig van *Scratch*

Oktober 2014: Inhandiging van navorsing

1.9 SAMEVATTING

Leerders in Noordwesprovinsie leer in graad 10 programmering aan deur middel van ŉ VP, *Scratch*, en skakel in graad 11 oor na programmering in ŉ SOOP, *Delphi*. Onderrig in ŉ VP verminder in ŉ mate die kompleksiteit van programmering, aangesien onderrig dadelik op strategiese kennis van programmering kan fokus, en nie op sintaktiese en konseptuele kennis soos in ŉ SOOP die geval is nie (sien 1.2). Dit kan nie as vanselfsprekend ervaar word dat programmeringsbegrippe vanaf *Scratch* na *Delphi* oorgedra gaan word nie. Onderrig moet pertinent op die oordrag van programmeringsbegrippe fokus (Carver, 1986:12). Navorsing is nie voorheen gedoen om te bepaal hoe om *Scratch* te onderrig met die oog op die oorgang na *Delphi* nie. Onderwysers moes egter reeds in 2012 *Scratch* aan graad 10 leerders

onderrig sodat leerders in graad 11 na gevorderde programmering in *Delphi* kan oorskakel (sien tabel 1.1). Met hierdie studie word dus gepoog om ’n bydrae te lewer ten einde riglyne te verskaf vir die onderrig van *Scratch* om die aanleer van programmering in *Delphi* te vergemaklik.

HOOFSTUK 2:

SCRATCH EN *DELPHI* AS PROGRAMMEERTALE TYDENS DIE ONDERRIG VAN PROGRAMMERING AAN IT-LEERDERS

2.1 INLEIDING

Die doel met die huidige navorsing was om te bepaal hoe *Scratch* as ’n visuele programmeertaal (VP) aan graad 10-leerders[1] onderrig behoort te word om die oorgang na *Delphi* as sintaksisgebaseerde objekgeoriënteerde programmeertaal (SOOP) te vergemaklik.

Ten einde die doel te bereik, word die volgende subdoelwitte van die studie in hierdie hoofstuk bespreek, naamlik eerstens om die aard van *Scratch* as ’n VP en *Delphi* as ’n SOOP afsonderlik te bepaal sowel as om aan te dui watter algemene programmeringsbeginsels reeds tydens die onderrig van *Scratch* vasgelê kan word. Laastens word daar spesifiek op die onderrig van *Scratch* en die voordele en nadele daarvan gefokus, om algemene programmeringsbeginsels wat sinvol daarmee onderrig kan word te identifiseer.

Voordat *Scratch* en *Delphi* individueel ondersoek word, word daar eers na die aard van programmering in die algemeen gekyk. Nadat ’n ondersoek gedoen is oor die aard van *Scratch* en *Delphi*, word die verskille en ooreenkomste tussen die twee tale in oënskou geneem en afleidings word gemaak oor programmeringsbeginsels en -begrippe wat reeds in *Scratch* vasgelê kan word. Daarna word die onderrig van programmering, en verskeie onderrig–leerstrategieë vir die effektiewe onderrig en leer van programmering bespreek. Aangesien daar twee verskillende benaderings bestaan waaruit objekgeoriënteerde programmering onderrig kan word, naamlik objekte-eerste of objekte-laaste, word daar ook gekyk na wat hierdie benaderings behels en watter uitwerking die gekose benadering op die onderrig van programmering het.

[1] Leerders in ’n hoërskool

2.2 BEGRIPSVERKLARING VAN TERME WAT IN DIE NAVORSING VOORKOM

'n *Programmeertaal* is die taal waarin instruksies aan 'n rekenaar gegee word om bepaalde aksies uit te voer. Die programmeertaal *Scratch* staan as 'n *visuele programmeertaal* (VP) bekend (Maloney *et al*., 2010:1). Met 'n VP word programmeringskode nie ingetik nie, maar blokke met ingeboude kode word deur middel van 'n muis saamgevoeg om programmeringskode op te bou.

'n *Objekgeoriënteerde programmeertaal* (OOP) is 'n programmeertaal waar programmering vanuit 'n objekgeoriënteerde benadering geskied. 'n *sintaksisgebaseerde objekgeoriënteerde programmeertaal* (SOOP) word gedefinieer as 'n programmeertaal waar 'n objekgeoriënteerde benadering (sien 2.5.2) gevolg word en programmeringskode volgens sintaktiese en semantiese reëls van die taal ingetik word (Kelleher & Pausch, 2005:86). Om die rede, word *Delphi* en *Java*, as SOOP'e beskou.

Die meeste programmeertale gebruik 'n *geïntegreerde ontwikkelingsomgewing* (GOO), naamlik die koppelvlak tussen die programmeerder en die programmeertaal, waar die ontwerp van programme plaasvind. Alhoewel programmeertale ooreenkomste bevat, is die benaderings waaruit geprogrammeer word, volgens Sebesta (2012:41), nie by almal dieselfde nie. Programmeertale word in die algemeen onder vier hoofkategorieë geklassifiseer, naamlik imperatief, funksioneel, logies en objekgeoriënteerd (Sebesta, 2012:41). OOP'e het vanuit *imperatiewe programmeertale*, soos *Pascal,* ontwikkel. By *imperatiewe programmeertale* word 'n prosedurele benadering gevolg, tenoor 'n objekgeoriënteerde benadering *wat* by OOP'e gevolg word (Sebesta, 2012:41). Hierdie benaderings word in 2.5.2 meer breedvoerig bespreek.

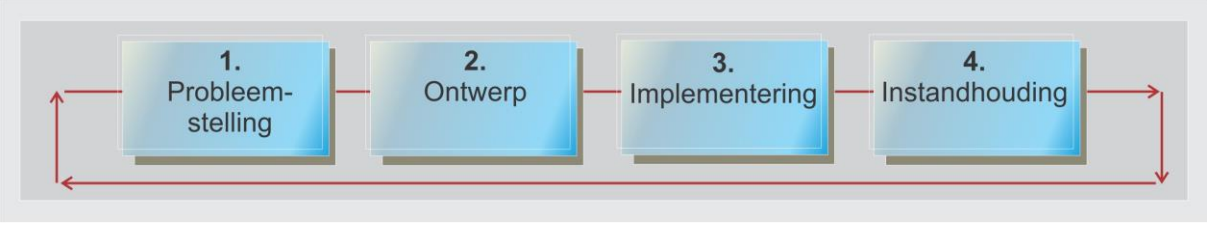
SOOP'e word in die bedryf gebruik om programme en stelsels te ontwikkel, maar word ook op skoolvlak in grade 11 en 12 gebruik, om programmering in die vak Inligtingstegnologie te onderrig. In Suid-Afrika bepaal die provinsiale onderwysowerheid die programmeertaal, naamlik *Java* of *Delphi*, wat tydens die onderrig van programmering in die vak Inligtingstegnologie in grade 11 en 12 gebruik moet word. Noordwesprovinsie het verkies om programmering in *Delphi* te onderrig.

Vervolgens word die aard van programmering van nader beskou.

2.3 DIE AARD VAN PROGRAMMERING

Programmering kan beskou word as ’n volledige proses wat by ’n probleem in die werklike lewe begin en met ’n geïmplementeerde rekenaarprogram eindig (Rogalski & Samurcay, 2010:6). Edsger Dijkstra, wat as ’n pionier in programmering beskou word (Misa, 2010:41), het gesê dat die intellektuele uitdagings van programmering groter is as die intellektuele uitdagings van teoretiese fisika. Dijkstra (1989:1392) verduidelik die kompleksiteit van programmering deur ’n program as ’n formule te beskryf. Die programmeerder se taak is om hierdie formule uit te dink en ’n reeks simbole vir die doel te manipuleer (Dijkstra, 1989:1392; Kelleher & Pausch, 2005:83). Dijkstra (1989:1392) beweer verder dat ’n programmeerder in konseptuele hiërargieë moet dink, dieper as wat die menslike verstand nog ooit moes doen. Alhoewel Dijkstra hierdie stellings ’n hele paar jaar gelede gemaak het en programmeertale intussen ontwikkel het om tred te hou met die veranderde rekenaaromgewing, het die basiese beginsels en begrippe van programmering sowel as die aard van programmering oor die afgelope 50 jaar baie dieselfde gebly (Sebesta, 2012:38) en word programmering steeds as ’n aktiwiteit beskou wat komplekse ontwerp behels (Rogalski & Samurcay, 2010:6). Volgens Donchev en Todorova (2013:84) is programmering ’n kreatiewe proses vir die uitdrukking van abstrakte idees en ’n komplekse kombinasie van wetenskap en kuns.

Alhoewel daar verskeie menings oor die aard van programmering bestaan, beweer Rogalski en Samurcay (2010:8) dat die proses van programmering vir enige programmeertaal as ’n sikliese proses voorgestel kan word (sien figuur 2.1), bestaande uit probleemvoorstelling, ontwerp, implementering en instandhouding.



Figuur 2.1: Die proses van programmering (Rogalski & Samurcay, 2010:8)

Die bostaande stappe kan almal onder een van die vyf domeine van programmering soos deur Du Boulay (1989) geïdentifiseer (verwysing in Sajaniemi & Kuittinen 2008:76; Bennedsen & Caspersen, 2008:9) en gekategoriseer word. Dié domeine is notasie, notasie-masjien, oriëntasie, strukture en pragmatiek. ’n Verduideliking van hierdie domeine word vervolgens verskaf.

Notasie

Notasie kan as die sintaksis en semantiek van ’n programmeertaal beskryf word (Kelleher & Pausch, 2005:83).

Notasie-masjien

’n Programmeerder moet verstaan hoe die notasie-masjien onderliggend aan die taal werk ten einde begrippe van ’n programmeertaal te verstaan, en te begryp hoe ’n program werk wat in ’n spesifieke programmeertaal geskryf is (Rogalski & Samurcay, 2010:12; Sajaniemi & Kuittinen, 2008:79). Die meeste programmeertale is gebaseer op die Von Neumann-rekenaarargitektuur en bevat bepaalde strukture soos herhaling- en besluitnemingstrukture en maak gebruik van veranderlikes om waardes tydelik te stoor (Sebesta, 2012:38). Die notasie-masjien is ’n onderliggende verduideliking van hoe begrippe soos byvoorbeeld veranderlikes, toevoer, afvoer en uitvoering van programme in ’n spesifieke programmeertaal werk (Sajaniemi & Kuittinen, 2008:79).

Oriëntasie

Die oriëntasie hou verband met die spesifieke benadering waaruit geprogrammeer word. Wanneer vanuit ’n prosedurele benadering geprogrammeer word, sal die oriëntasie wees om programme te skryf wat betekenis het, in die sin dat dit betekenisvolle aksies uitvoer en betekenisvolle bewerkings doen. Wanneer daar egter vanuit ’n objekgeoriënteerde benadering geprogrammeer word, is die oriëntasie om eers konseptuele modelle vir data te skep en later die funksionaliteit van die program te oordink (Sajaniemi & Kuittinen, 2008:76).

Strukture

Sajaniemi en Kuittinen (2008:79) beskryf strukture as abstrakte oplossings vir probleme. Volgens Rogalski en Samurcay (2010:12) bestaan daar verskeie strukture in programmering, naamlik datastrukture (soos rekords en skikkings), programmeringsbegrippe (soos funksies en veranderlikes), herhaling- en beheerstrukture, en algoritmes. Programmeertale benodig volgens Sebesta (2012:368) twee soorte beheerstrukture om algoritmes en probleme suksesvol te implementeer, naamlik 'n struktuur om te kan besluit watter pad van uitvoering gevolg moet word en 'n struktuur om herhalings te beheer. In hierdie navorsing word na eersgenoemde verwys as *besluitnemingstrukture* en na laasgenoemde as *herhalingstrukture*.

Alhoewel bepaalde programmeringsbegrippe, soos hierbo genoem, soos 'n goue draad deur programmeertale loop, is daar duidelike sintaktiese verskille in die notasie van verskillende programmeertale (Kelleher & Pausch, 2005:83). Die oplossing van probleme kan, ongeag die sintaktiese verskille van programmeertale, universeel deur middel van algoritmes voorgestel word. 'n Algoritme kan beskryf word as 'n stap-vir-stap voorstelling vir die oplossing van 'n bepaalde probleem (Kerman, 2002:62), en die programmeertaal kan gesien word as die gereedskap waarmee die algoritme geïmplementeer word (Gomes & Mendes, 2007:1). Wanneer 'n algoritme in bepaalde programmeringskode geïmplementeer word, moet die programmeerder vertroud wees met die bepaalde notasie en notasie-masjien van die taal (Sajaniemi & Kuittinen, 2008:77). Algoritmes kan op verskeie maniere voorgestel word, naamlik deur middel van pseudokode, wat 'n gestruktureerde, woordelikse beskrywing van die probleem is, of deur middel van grafiese voorstellings, soos byvoorbeeld vloeidiagramme (Kerman, 2002:62–66; Sebesta, 2012:368). Om 'n algoritme te ontwerp moet 'n programmeerder abstrakte begrippe, soos beheerstrukture, verstaan en dit toepas om oplossings vir konkrete probleme te ontwerp (Gomes & Mendes, 2007:1). Die ontwerp van 'n algoritme word algemeen beskou as een van die moeilikste stappe van die programmeringsproses (Gomes & Mendes, 2007:1; Jenkins, 2002:5).

Pragmatiek

Pragmatiek behels die vaardighede om programme te beplan, dit te ontwikkel, te toets en te ontfout (Bennedsen & Caspersen, 2008:9) en word ook die proses van programmering genoem (Sajaniemi & Kuittinen, 2008:76). Hierdie vaardighede behels probleemoplossing, wat onder andere begrip van die probleem self impliseer, die oordra van verwante kennis uit vorige ervarings na nuwe probleme, die vermoë om oor die probleem en die oplossing te kan reflekteer, en om die deursettingsvermoë te hê en aan te hou totdat die probleem opgelos is (Gomes & Mendes, 2007:2). Jenkins (2002:55) beskryf pragmatiek as die uitvoering van meervoudige prosesse soos die ontleding van ’n konkrete probleem en toepassing van abstrakte begrippe om algoritmes te ontwerp.

Uit die voorafgaande is dit duidelik dat die aard van programmering kompleks is en dat dit uit ingewikkelde prosesse en komplekse begrippe bestaan (Bennedsen & Caspersen, 2008:9; Jenkins, 2002:55; Sajaniemi & Kuittinen, 2008:76). Vervolgens word die twee programmeertale, *Scratch* en *Delphi*, wat tans tydens die onderrig van Inligtingstegnologie in Noordwesprovinsie gebruik word, van nader beskou met die doel om die onderrig van *Scratch* met die oog op die oorgang na *Delphi* te ondersoek.

2.4 *SCRATCH* AS ’N VISUELE PROGRAMMEERTAAL

Scratch is geïnspireer deur hip-hop-platejoggies wat langspeelplate gekrap het en kombinasies van liedjies saamgevoeg het om hulle eie unieke musiek te skep (Ford, 2009:xiv). Net so laat *Scratch* leerders toe om hulle eie unieke programme te ontwerp deur van *Sprites*, kostuums, kodeblokke en klanke gebruik te maak. *Scratch* kan dus as ’n media-ryke programmeringsomgewing (Malan & Leitner, 2007:223) beskryf word. Dit was aanvanklik op leerders tussen die ouderdomme 8 en 16 jaar gerig, alhoewel dit deesdae deur mense van alle ouderdomme gebruik word (Resnick *et al*., 2009:60). Sleutelbordvaardighede is nie belangrik nie en programmeringskode hoef nie aangeleer te word nie, aangesien programmering geskied deur die insleep van blokke met ingeboude programmeringskode (Malan & Leitner, 2007:223).

2.4.1 Die doel van *Scratch*

Scratch is aanvanklik in 2003 deur MIT Media Lab se Lifelong Kindergarden Group, onder leiding van Mitchel Resnick (MIT Media Lab, 2003) ontwikkel. Die doel was om programmering aan mense van alle ouderdomme en uit alle agtergronde bekend te stel (Resnick *et al.*, 2009:60), om hulle te motiveer om te programmeer (Wolz *et al.*, 2008:298), sistematies te leer dink, hulle kreatiewe denke te ontwikkel en saam met ander te leer werk (MIT Media Lab, 2003). Hierdie vaardighede word deur MIT as noodsaaklike lewensvaardighede beskou wat vir die 21ste eeu benodig word (MIT Media Lab, 2003). Volgens Resnick *et al.* (2009:60) was die doel met die ontwikkeling van *Scratch* nie om van almal professionele programmeerders te maak nie, maar om ’n generasie van kreatiewe, sistematiese denkers op te lei, wat hulle idees deur middel van programmering kan uitdruk. Hierdie idees behels ’n wye verskeidenheid projekte (Malan & Leitner, 2007:224) soos interaktiewe stories, verjaardagkaartjies, speletjies en die uitbeelding van aktuele kwessies in die samelewing. Daar is al soveel *Scratch*-projekte op die *Scratch*-webtuiste gelaai dat dit as die ‘YouTube van interaktiewe media’ bekend staan (Resnick *et al.*, 2009:60). Die doel met *Scratch* was dus aanvanklik nie om dit as aanvangsprogrammeertaal in skole te gebruik nie, alhoewel dit in toenemende mate in skole gebruik word (Maloney *et al.*, 2010:2).

2.4.2 Die aard van *Scratch*

Die ontwikkelaars van *Scratch* het ’n beleid van lae vloer, wye mure en hoë dak by die ontwerp van *Scratch* toegepas (Malan & Leitner, 2007:223; Resnick *et al*., 2009:63). Die *lae vloer* verwys daarna dat beginnerprogrammeerders met *Scratch* bemagtig word om reeds met hulle eerste kennismaking programme te kan skryf wat animasie of speletjies kan uitvoer (Malan & Leitner, 2007:223) en ’n projek kan skep sonder enige kennis van ander programmeertale (Ford 2009:15; Malan & Leitner, 2007:225). Die *wye mure* verwys na die veelsydigheid van *Scratch*, naamlik dat dit moontlik moet wees om ’n groot verskeidenheid soorte projekte daarmee te skep (Malan & Leitner, 2007:224). Die *hoë dak* verwys daarna dat gevorderde gebruikers meer komplekse programme moet kan ontwikkel (Resnick *et al*., 2009:63).

Die aard van *Scratch* word met die volgende drie kernbeginsels in die ontwerp daarvan saamgevat, naamlik *"*more tinkerable, more meaningful and more social" (Resnick *et al.*, 2009:63). Hierdie drie kernbeginsels word vervolgens verduidelik.

2.4.2.1 Leer deur te speel (tinkerable)

Die ontwikkelaars van *Scratch* het soos 'n kind met boublokkies speel, dit speel-speel inmekaar sit en nuwe idees kry, te werk gegaan (Resnick *et al.*, 2009:60). *Scratch* kan as 'n leer-deur-te-speel-programmeertaal beskryf word, omdat kodeblokke inmekaar gepas word, leerders dan die gevolg daarvan kan sien as programme uitvoer, en nuwe idees daaruit verkry. Hierdie programmeringsomgewing moedig leerders aan om *Scratch* op 'n eksploratiewe en eksperimentele manier te gebruik (Meerbaum-Salant *et al.,* 2013:240; Resnick *et al.*, 2009:60; Wolz *et al.*, 2008:298). Programme word op 'n iteratiewe, inkrementele manier ontwerp deur eers 'n paar blokkies inmekaar te pas, die program uit te voer, te kyk wat gebeur, blokkies by te voeg of weg te neem, te kyk wat gebeur, en so met die proses aan te hou totdat die leerder tevrede is dat die program reg werk. Hierdie werkswyse word moontlik gemaak deur die interaktiewe aard van *Scratch* waar veranderings onmiddellik waargeneem kan word (Resnick *et al.*, 2009:60).

2.4.2.2 Betekenisvol

Volgens Resnick *et al.* (2009:64) leer 'n persoon die beste as daar aan betekenisvolle projekte gewerk word. *Scratch* is juis so ontwerp, aangesien leerders aan 'n verskeidenheid projekte wat hulle interesseer en wat met hulle eie wêreld verband hou, kan werk. Hulle kan byvoorbeeld stories, speletjies en oplossings vir probleme in ander vakgebiede ontwerp. Hulle kan ook projekte verpersoonlik, deur hulle eie klankopnames, prente en foto's in te voeg.

2.4.2.3 Sosiaal

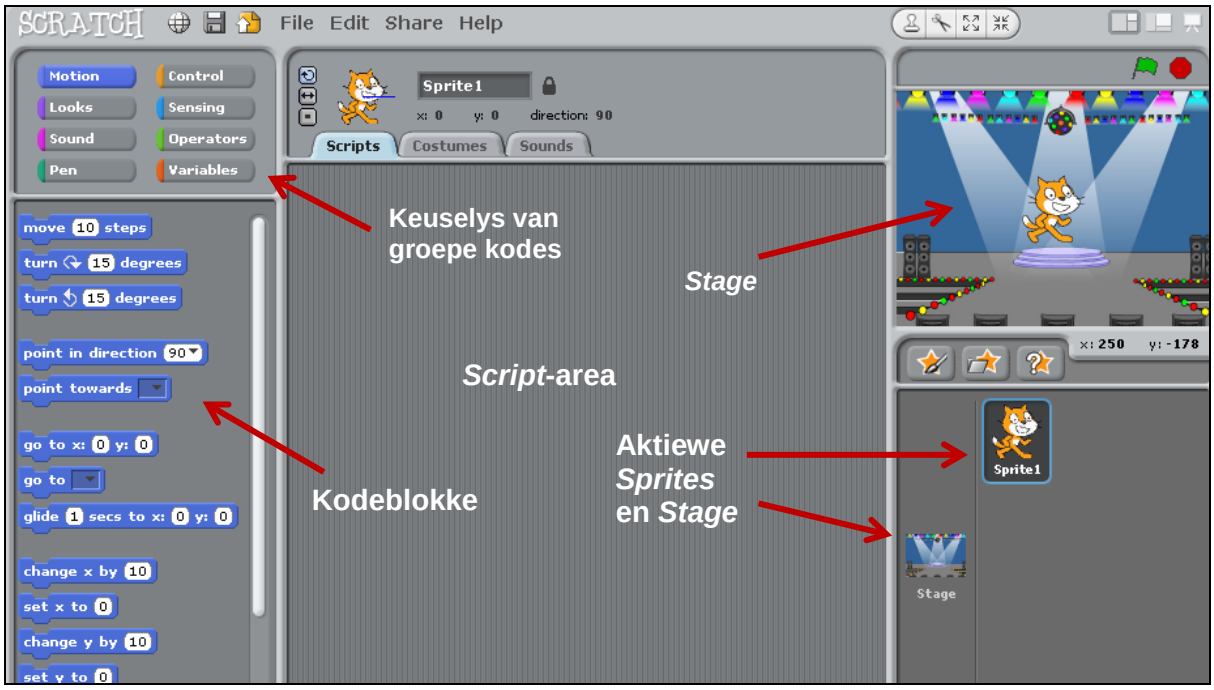
'n Persoon wat in *Scratch* programmeer, word algemeen 'n *Scratch*er genoem en is deel van die aanlyn, sosiale gemeenskap, waar *Scratch*ers van enige ouderdom hulle ervarings, oplossings en probleme met mekaar deel. Programmeerders skryf in die meeste gevalle programme vir ander mense en wil, net soos kunstenaars, hulle werk met ander deel en 'n gehoor hê wat hulle werk kan waardeer (Resnick *et al.*,

2009:65). ’n Deel van die genot van *Scratch* is dus om ’n voltooide *Scratch*-projek op die Internet te plaas, dit met ander gebruikers te deel en so deel te word van die wêreldwye *Scratch*-gemeenskap (Ford, 2009:5). Alhoewel *Scratchers* algemeen onder skuilname skryf en projekte oplaai, is ’n mens deurgaans bewus van die samehorigheidsgevoel en opwinding wat onder *Scratchers* heers (MIT Media Lab, 2007). *Scratchers* voer mekaar se projekte uit, soek deur die projekte na interessante idees en lewer kommentaar op die projekte. Hierdie reaksie van ander *Scratch*-gebruikers oor die wêreld heen kan leerders aanspoor om aan te hou programmeer en hulle selfvertroue te verhoog, omdat hulle voel dat ander gebruikers hulle projekte waardeer (Ford, 2009:5).

Die *Scratch*-webtuiste gee ook aandag aan sosiale opvoeding, omdat respek vir ander persone en opbouende kommentaar ’n voorvereiste is (MIT Media Lab, 2007). Vaardighede soos koöperatiewe probleemoplossing (Resnick *et al.*, 2009:60,65), groep-interaksie en kommunikasievaardighede word op ’n natuurlike manier beoefen deur interaksie met die *Scratch*-aanlyngemeenskap (Wolz *et al.,* 2008:299). *Scratch* bring deur middel van hierdie aanlyngemeenskap dus ’n sosiale dimensie na programmering wat kommunikasie en samewerking tussen programmeerders aanmoedig (Wolz *et al.*, 2008:298).

2.4.3 Die *Scratch*-programmeringsomgewing

Die *Scratch*-programmeringsomgewing herinner baie aan ’n toneelopvoering, waar die hoofrolle deur karakters wat *Sprites* genoem word, gespeel word. Alle instruksies wat deur programmeerders gegee word, het ten doel om *Sprites* te programmeer om sekere take (wat wissel van sing en dans tot komplekse wiskundige bewerkings) uit te voer, met die oog op die oplossing van ’n bepaalde probleem. Die tonele speel af op ’n verhoog, wat ’n *Stage* genoem word. Die opstelling van die *Stage* bepaal hoe die agtergrond van die area lyk waarbinne die *Sprites* beweeg (MIT Media Lab, 2003). *Sprites* word deur middel van *Scripts* beheer, wat instruksies is wat saamgestel word deur kodeblokke inmekaar te pas (MIT Media Lab, 2014b).



Figuur 2.2: Die *Scratch*-geïntegreerde ontwikkelingsomgewing (GOO)

Die *Scratch*-GOO is ’n grafiese koppelvlak wat uit ’n kodeblok-area, ’n *Script*-area wat kodeblokke bevat wat ingesleep is, ’n *Sprite*-area, die verhoog (*Stage*), en ’n area wat *Sprites* vertoon wat in die spesifieke program optree (sien figuur 2.2). Die GOO is so ontwerp dat al vier dele en al die blokke wat aan ’n bepaalde groep kodes behoort sigbaar is, ter wille van eenvoudige ontwerp sodat die leerder al die komponente wat moontlik gebruik kan word, kan sien (Maloney *et al.*, 2010:7).

2.4.4 *Scratch* as programmeertaal

Volgens Ford (2009:7) bevat *Scratch* al die basiese begrippe wat in ander programmeertale gebruik word. Alhoewel die *Scratch*-omgewing ’n mens aan ’n speletjie herinner, is dit ’n kragtige programmeertaal waarmee gebruikers ’n verskeidenheid toepassings kan ontwikkel (Ford, 2009:xv) en neem dit tyd om die tegniese aspekte daarvan sowel as die programmeringsbeginsels wat daarin bevat is, aan te leer (Meerbaum-Salant *et al.,* 2013:263). *Scratch* bevat egter nie al die programmeringseienskappe wat in gevorderde programmeertale ingesluit is nie soos byvoorbeeld metodes, data-strukture, datatipes, oorerflikheid, terugstuur van waardes en polimorfisme (Malan & Leitner, 2007:227). *Scratch* fokus slegs op die

aanleer van begrippe van programmering, en volgens Malan en Leitner (2007:227) skep dit ’n stewige grondslag waarop leerders kan bou met die oorgang na meer gevorderde programmeertale. Dit is dan ook juis om die rede waarom die Departement van Basiese Onderwys (DBO) op die gebruik van *Scratch* as programmeertaal in graad 10, waar leerders vir die eerste keer met programmeringsvaardighede kennis maak, besluit het (DBE, 2011:11).

2.4.4.1 Scratch as ’n objekgeoriënteerde programmeertaal

Alhoewel *Scratch* nie as ’n volwaardige OOP gesien kan word nie (Malan & Leitner, 2007:227) kan *Sprites* as objekte gesien word, omdat hulle oor eienskappe (soos byvoorbeeld X-koördinate, Y-koördinate, rigting) en gedrag (kan aksies uitvoer) beskik (Harvey & Mönig, 2010:8; Maloney *et al*., 2010:10). *Scratch* bevat ook toepassing van enkapsulasie op veranderlikes en *Script*s, aangesien instruksies slegs in die *Sprites* waarin dit voorkom uitgevoer kan word, en *Sprites* nie toegang het tot veranderlikes in ander *Scripts* nie, tensy dit op die *Stage* geplaas word (Maloney *et al*., 2010:10). *Scratch* beskik oor ’n baie eenvoudige manier om boodskappe aan ander *Scripts* te stuur, naamlik *Sprites* wat boodskappe na alle ander *Sprites* uitsaai (Maloney *et al*., 2010:11). In die uitsaaikode kan daar nie argumente of veranderlikes wees nie. Prosedures kan nie met *Scratch* geskryf word nie, maar *Sprites* kan wel gedeel word en dit moedig hergebruik van kode aan (Maloney *et al*., 2010:11). Alhoewel *Scratch* dus sommige begrippe van ’n OOP bevat, sluit dit nie oorerwing of polimorfisme in nie en gevolglik beweer Maloney *et al*. (2010:10) dat *Scratch* slegs as ’n objekgebaseerde taal geklassifiseer kan word en nie as ’n ware OOP nie.

Vervolgens word daar na sommige aspekte van die taal van *Scratch* gekyk.

2.4.4.2 Die taal van Scratch

Programinstruksies is op gekleurde blokke, wat spesifieke vorms het, ingebou. Op elke blok staan ’n beskrywing van die opdrag wat die bepaalde blok aan ’n *Sprite* gee. Hierdie boublokkie-benadering onderskei *Scratch* van ander programmeertale aangesien *Scratch* maklik aangeleer kan word en die logika selfs gevolg kan word deur persone wat geen programmeringsagtergrond het nie (Ford, 2009:xv).

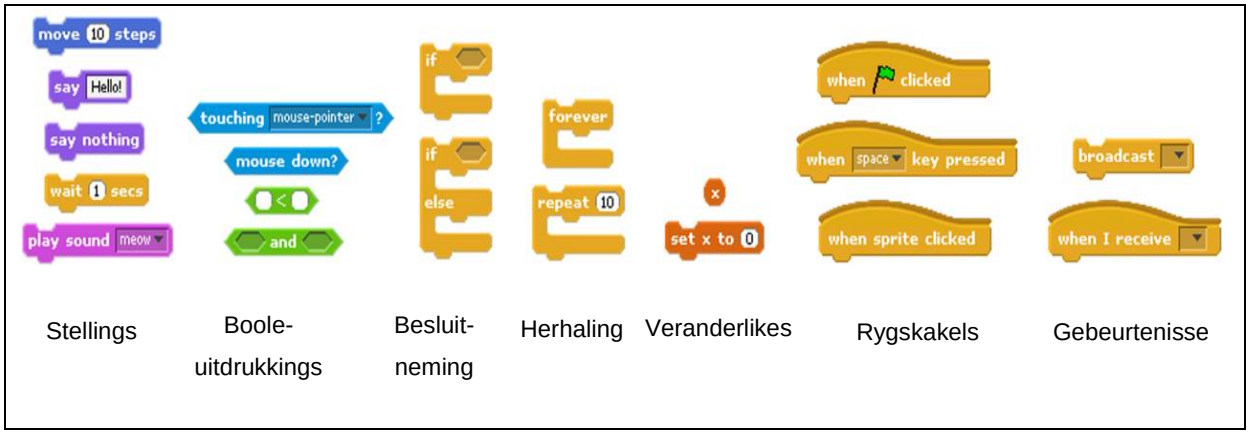
Tabel 2.1: Soorte *Scratch*-blokke

1. Stapelblok	
2. Hoedblok	
3. Funksieblok	
4. Kontrole-/beheerblokke	

Ford (2009:55) deel kodeblokke in drie hoofgroepe in, naamlik stapelblokke, hoedblokke en funksieblokke (sien tabel 2.1). Maloney *et al.* (2010:8) klassifiseer die kontrole- of beheerblokke as ’n vierde groep blokke (sien tabel 2.1). Die stapelblokke het ’n inham aan die bo-kant of ’n uitham aan die onderkant om aan te dui hoe die blokke inmekaar pas, en word gebruik om instruksies aan *Sprites* te gee. ’n Hoedblok het ’n ronding bo en ’n uitham aan die onderkant en word bo-aan stapelblokke geplaas, met die doel om aan te dui dat die *Script* sal uitvoer as ’n bepaalde gebeurtenis plaasvind. Alle *Script*s wat met die hoedblok met ’n groen vlaggie op begin, sal byvoorbeeld uitvoer indien die gebruiker op die *Start*-opsie klik (Maloney *et al.*, 2010:8). ’n Funksieblok het ’n gladde bokant en onderkant en verskaf toevoer of inligting aan ander blokke. Hierdie soort blokke kan argumente vir ander blokke genoem word en die posisies waarin dit inpas, word parameter-gleuwe genoem (Maloney *et al.*, 2010:8). Funksieblokke kan ook inmekaar ge-nes word, om wiskundige en ander uitdrukkings op te bou. Kontrole- of beheerblokke het tipies ’n C-vorm en omsluit die instruksies wat dit beheer.

Die vier hoofgroepe blokke word in agt kleurgroepe verdeel, wat elk weer ’n groep instruksies verteenwoordig wat in programmering gebruik word, naamlik beweging (donkerblou), voorkoms (pers), klank (pienk), teken (donkergroen), beheer (oranje), sintuiglike waarneming (ligblou), bewerkings (liggroen) en veranderlikes (rooi) (Malan

& Leitner, 2007:224). Die agt groepe blokke bestaan in totaal uit meer as ’n 100 verskillende blokke (Ford, 2009:60) wat elk ’n bepaalde funksie het. Malan en Leitner (2007:226) het ’n indeling van blokke volgens programmeringsbegrippe gemaak en het die volgende groepe geïdentifiseer: stellings, Boole-uitdrukkings, besluitneming, herhaling, veranderlikes, rygskakels en gebeurtenisse. Voorbeelde van kodeblokke wat onder hierdie kategorieë geklassifiseer is, word in figuur 2.3 aangetoon.



Figuur 2.3: Kodeblokke volgens programmeringsbegrippe (Malan & Leitner, 2007:226)

Op elke kodeblok staan ’n beskrywing van die instruksie wat die bepaalde blok aan ’n *Sprite* gee. Indien daar op ’n blok regsgeklik word, is ’n hulp-opsie wat ’n verduideliking gee van die blok se funksie, hoe dit gebruik kan word, asook ’n voorbeeld van hoe die kode in konteks gebruik kan word, dadelik beskikbaar.

2.4.4.3 Sintaksis

Die sintaksis van *Scratch* word visueel uitgebeeld in die vorms van die kodeblokke en die manier waarop die blokke inmekaar pas (Maloney *et al.*, 2010:7). Wanneer ’n blok ingesleep word, vertoon *Scratch* die moontlike passings deur die rand van sodanige blokke wit te verlig. Hierdie vasgestelde kombinasies van blokke het die voordeel dat ’n leerder op die oplossing en logika van programmering kan fokus, omdat geen sintaksisfoute gemaak kan word nie (Malan & Leitner, 2007:223). Dit is juis hierdie sintaksisfoute wat herhaaldelik in ’n SOOP voorkom en wat gedurig deur programmeerders gekorrigeer moet word (Ford, 2009:7).

2.4.4.4 Veranderlikes en datatipes

Net soos ander programmeertale, stoor *Scratch* data en antwoorde op bewerkings deur van veranderlikes gebruik te maak (Maloney *et al*., 2010:6). Leerders verstaan die begrip van veranderlikes baie makliker in *Scratch* as in gewone programmeertale (Malan & Leitner, 2007:226), omdat dit deur middel van monitors konkreet en visueel voorgestel word (Maloney *et al*., 2010:6) (sien tabel 2.2).

Tabel 2.2: Voorstelling van veranderlikes in *Scratch*

'n Monitor wat die inhoud van die veranderlike **Som** aandui.	
Instruksie om **X** se waarde na 0 te verander.	

By veranderlikes is veral twee aspekte van belang, naamlik reikwydte en die benoeming daarvan (Ford, 2009:143). 'n Veranderlike het *lokale reikwydte* wanneer dit aan 'n spesifieke *Sprite* behoort, naamlik die *Sprite* wat aktief was toe die veranderlike gedefinieer is en die waarde van die veranderlike kan slegs deur hierdie *Sprite* verander word. 'n Veranderlike kan egter deur al die *Sprites* verander word indien dit op die *Stage* geplaas word en dan het die veranderlike *globale reikwydte*. 'n Tweede aspek is die benoeming van veranderlike name. *Scratch* is baie buigsaam wat die benoeming van veranderlikes betref – 'n veranderlike naam kan uit enige kombinasie van karakters bestaan en kan enige lengte wees (Ford, 2009:143).

In *Scratch* word nie pertinent genoem dat daar verskillende datatipes aan veranderlikes toegeken word nie. Daar word wel onderskeid gemaak tussen getalle, teks en Boole-waardes, maar slegs deur middel van die parameter-gleuwe se vorms (Maloney *et al*., 2010:8), soos aangedui in tabel 2.3. Outomatiese omskakeling tussen getal- en tekstipes vind tydens afvoer plaas, sonder dat die leerder daarvan bewus is.

Tabel 2.3: Die voorstelling van datatipes in *Scratch*

Boole-waardes word deur parameter gleuwe met hoekige kante voorgestel	
Numeriese waardes word deur parameter gleuwe met ronde kante voorgestel	
Tekswaardes word deur reghoekige parameter gleuwe voorgestel	

2.4.4.5 Herhaling- en besluitnemingstrukture

Die begrippe *herhaling* en *voorwaardelike uitvoering* word visueel in *Scratch* voorgestel en volg so natuurlik uit die bewegings van *Sprites* dat leerders dit vinnig begryp en in programme kan gebruik (Malan & Leitner, 2007:226). Herhalingstrukture in *Scratch* kan as onvoorwaardelik of voorwaardelik geklassifiseer word (Ford, 2009:180–184). Die onvoorwaardelike herhalingstrukture is FOREVER, wat sal aanhou totdat die gebruiker die rooi stop-alles-opsie klik, en REPEAT-N, wat slegs ’n gespesifiseerde aantal herhalings sal uitvoer. Die voorwaardelike herhalingstrukture is FOREVER-IF en REPEAT-UNTIL. By hierdie herhalingstrukture moet ’n Boole-voorwaarde as deel van die kode gespesifiseer word, wat dan sal bepaal of die uitvoering van die blokke wat deur die lusstruktuur omarm word, moet voortgaan (Ford, 2009:184).

Scratch bevat besluitnemingstrukture IF en IF-ELSE (Ford, 2009:184) wat soortgelyk is aan keusestrukture wat in ander programmeertale aangetref word. Dit bevat egter nie ’n ekwivalent vir die CASE-stelling in *Delphi* nie.

2.4.4.6 Uitvoering van programme

Om ’n program in *Scratch* uit te voer, moet ’n blok geklik word en die daaropvolgende blokke sal sekwensieel, van bo na onder, uitgevoer word. Dit is selfs moontlik om net sekere blokke uit te voer (Maloney *et al*., 2010:4). Die tempo van uitvoering kan stadiger gestel word sodat ’n leerder reël vir reël kan waarneem watter blok uitgevoer word. *Scratch* beskik oor ’n fasiliteit wat, deur een instruksie (reël) van die program op

’n keer na te gaan, foutopsporing vergemaklik. Elke reël wat uitgevoer word, word met ’n geel kleur verlig. Die uitvoering van ’n program in *Scratch* is dus nie ’n geheimsinnige gebeurtenis wat agter die skerms plaasvind nie, maar ’n waarneembare, konkrete aksie (Maloney *et al*., 2010:3).

Scratch word ook ’n dinamiese programmeertaal genoem, omdat veranderings aan programme gemaak kan word terwyl die program besig is om uit te voer. Die programmeerder kan gevolglik onmiddellik die effek van die verandering waarneem (Maloney *et al*., 2010:4).

2.4.5 Programmering in *Scratch*

Omdat interaktiewe toepassings met *Scratch* geskep word, moet die programmeerder voortdurend in gedagte hou dat bepaalde gebeurtenisse kan plaasvind, of dat bepaalde scenario’s bestaan, soos byvoorbeeld dat *Sprites* aan mekaar kan raak, ’n sekere afstand van mekaar af is, of dat die gebruiker ’n bepaalde sleutel gedruk het (Ford, 2009:119). Alhoewel dit ook kan lyk of leerders speel wanneer hulle in *Scratch* programmeer, is die teendeel juis waar. Programmering in *Scratch* behels onder andere die aanleer van wiskundige begrippe en verskeie begrippe van rekenaarverwerking (Resnick *et al.*, 2009:60). Talle begrippe wat in ander programmeertale gebruik word, is ook in *Scratch* teenwoordig (Maloney *et al.*, 2010:14). Daarmee saam moet programmeerders in *Scratch* deur die hele proses van programmering werk (sien figuur 2.1 en figuur 2.9). Hierdie proses word vervolgens vanuit ’n *Scratch*-omgewing beskou.

2.4.5.1 Probleemvoorstelling en ontwerp

Gao (2011:1267) voer aan dat dit nie die sintaksis is wat die moeilik deel van programmering uitmaak nie, maar die oplos van probleme. Probleemoplossing is die kern van programmering en indien ’n leerder in *Scratch* programmeer, beteken dit nie dat probleemoplossing noodwendig makliker gaan wees nie. Die visuele omgewing van *Scratch* vereis dat leerders probleemoplossing anders as met ’n SOOP benader. Scratch bevat reeds die visuele blokke om ’n oplossing vir die probleem te bou en die uitwerking daarvan kan boonop onmiddellik gevolg word. Hierdie onmiddelike terugvoer verskaf ’n geleentheid om oor denke te reflekteer, omdat die probleemoplossingsproses konkreet voorgestel word (Resnick e*t al.*, 2009:60).

Maloney *et al.* (2010:8) voer aan dat *Scratch*-blokke as ’n soort pseudokode gebruik kan word, omdat leerders ná kennismaking met *Scratch* neig om aan *Scratch*-blokke te dink wanneer algoritmes ontwerp word.

Programmeerders in prosedurele programmeertale (Bergin, 2009:1) sowel as ervare SOOP-programmeerders (Détienne, 2010:57) gebruik gewoonlik ’n bo-na-onder-strategie om probleme op te los. Dit beteken dat ’n program as een groot probleem wat opgelos moet word, beskou word. Hierdie probleem word dan herhaaldelik in subprobleme onderverdeel, wat weer saamgevoeg word om die oorkoepelende probleem op te los (Bergin, 2009:1). Dit wil egter voorkom asof leerders in *Scratch* probleemoplossing vanuit ’n onder-na-bo-strategie aanpak (Maloney *et al*., 2010:4; Meerbaum-Salant *et al*., 2011:169), naamlik om met ’n paar blokke kode te begin, nog blokke by te voeg en dit in groter *Scripts* te kombineer, om ’n probleem op te los. Maloney *et al*. (2010:4) voer aan dat die leer-deur-te-speel-aard van *Scratch* (sien 2.4.2) daartoe lei dat leerders programme ontwerp deur van eksperimentering gebruik te maak en dus ’n onder-na-bo-strategie gebruik om probleme op te los. Détienne (2010:57) beweer egter dat ’n onder-na-bo-strategie die tipiese probleemoplossingstrategie van beginnerprogrammeerders is, ongeag of hulle uit ’n prosedurele of objekgeoriënteerde benadering programmeer.

2.4.5.2 Implementering

Die implementering van die oplossing behels dat die *Stage* opgestel word en kodeblokke in *Scratch* ingesleep word (Malan & Leitner, 2007:224). Hierdie samestelling van blokke verskaf dus ’n *Scratch*-programmeringskode vir die voorafopgestelde algoritme om die probleem op te los.

2.4.5.3 Instandhouding, fouthantering en toetsing

Foute in *Scratch*-programme word beperk deur die konfigurasie van blokke wat op mekaar pas (Ford, 2009:xiv), aangesien kodeblokke so ontwerp is dat ’n blok slegs in ’n ander blok kan pas as dit ’n logiese volgorde van instruksies voorstel. Selfs indien ’n waarde wat buite bepaalde grense val, ingevoer sou word, sou die *Scratch*-blokke steeds probeer om iets sinvols daarmee te doen eerder as dat die program ’n foutboodskap gee (Maloney *et al*., 2010:6). Dit is egter wel moontlik dat ’n *Scratch*-program verkeerd werk of ’n afvoerfout gee indien dit ’n ongeldige instruksie probeer

uitvoer, soos byvoorbeeld wanneer ’n bewerking uitgevoer word waar met nul gedeel word. In so ’n geval sal die program staak en die toepaslike blokke met rooi omlyn word (Maloney *et al.,* 2010:5). Programmeerders in *Scratch* kan ook steeds logiese foute maak, naamlik programme skryf wat wel uitvoer, maar die probleem verkeerd oplos. *Scratch*-programmeerders moet dus, net soos ander programmeerders, seker maak dat hulle programme die korrekte afvoer lewer deur behoorlik te toets dat ’n program onder alle omstandighede en met verskillende soorte data korrek sal funksioneer (Ford, 2009:278–281).

’n Nuttige fasiliteit in *Scratch* vir die opspoor van logiese foute is die gebruik van monitors (Maloney *et al.*, 2010:6) om die inhoud van veranderlike waardes te vertoon. Indien die monitors geaktiveer is, sal die waardes van die veranderlikes tydens uitvoering van die program vertoon word, sodat die programmeerder gedurig kan sien wat die inhoud van veranderlikes in die geheue is en of die program uitvoer soos wat die bedoeling was om die gegewe probleem op te los.

’n Verdere manier om te help met die opspoor van logiese foute is om net op een kodeblok te dubbelklik om daardie spesifieke gedeelte uit te voer. Indien ’n *Scratch*-program dus nie reg uitvoer nie, kan dit in klein deeltjies onderverdeel word en elke deel kan afsonderlik uitgevoer word om so te bepaal waar logiese foute voorkom (Maloney *et al*., 2010:5). Indien ’n leerder sukkel om ’n fout op te spoor, kan hy/sy op die help-kieslys klik en die *Scratch*-aanlynhulpfasiliteit sal geaktiveer word indien die rekenaar aan die Internet gekoppel is. As die fout steeds nie opgespoor kan word nie, kan hulp deur middel van die *Scratch*-aanlynforum by ander *Scratch*-programmeerders gesoek word (MIT Media Lab, 2007).

Uit die voorafgaande is dit duidelik dat die ontwikkelaars van *Scratch* maatreëls gereed het om te verseker dat foutopsporing nie ’n struikelblok in *Scratch*-programmering is nie. Swak programmeringsbeginsels, soos byvoorbeeld die onderverdeel van ’n *Script* in baie kleiner *Script*s en die gebruik van ongestruktureerde beheerstrukture, kan veroorsaak dat programme moeilik ontfout word. In sodanige gevalle kan dit so moeilik wees om die *Script*s te ontfout dat leerders moed opgee (Meerbaum-Salant *et al.*, 2011:170). In paragraaf 2.8.2.2 is hierdie nadele van *Scratch* volledig bespreek.

2.5 *DELPHI* AS ’N SINTAKSISGEBASEERDE OBJEKGEORIËNTEERDE PROGRAMMEERTAAL

In hierdie afdeling word die programmeertaal *Delphi* bespreek ten einde te bepaal watter ooreenkomste en verskille tussen *Scratch* en *Delphi* bestaan. Die doel en aard van *Delphi* word eerstens bespreek, en daarna word *Delphi* as sintaksisgebaseerde objekgeoriënteerde programmeertaal (SOOP) bespreek.

2.5.1 Die doel van *Delphi*

Delphi het uit die programmeertaal *Pascal* ontstaan en is in 1995 deur die Borland-maatskappy ontwikkel (Kerman, 2002:xv). Die doel met die ontwikkeling van *Pascal* in 1971 was aanvanklik dat dit as onderrigtaal vir inleidende programmering moes dien (Kerman, 2002:xv). Dit is egter later, weens die eenvoud en veelsydigheid van die taal, ook vir die ontwikkeling van kommersiële programme gebruik (Kerman, 2002:xv). Die ontwikkeling van ’n grafiese koppelvlak vir bedryfstelsels soos *Windows* het tot die gevolg gehad dat die GOO’s in programmeertale ook grafies moes wees. Dit, tesame met die koms van objekgeoriënteerde programmering, het daartoe gelei dat *Pascal* uitgebrei moes word en gevolglik is die programmeertaal *Delphi* ontwikkel (Kerman, 2002:xv). *Delphi* is aanvanklik ontwerp vir die ontwikkeling van programme in ’n *Windows*-omgewing, maar is as gevolg van die relatief eenvoudige sintaksis ook later as onderrigtaal vir aanvangsprogrammeerders gebruik (Kerman, 2002:xvi).

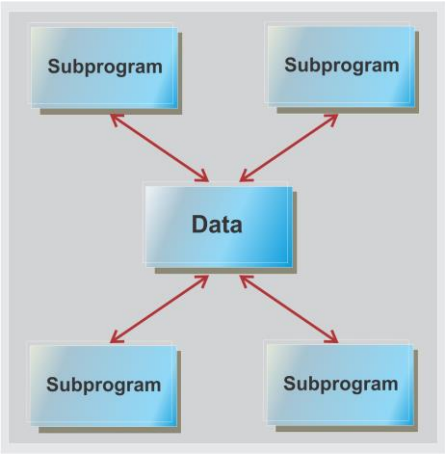
2.5.2 Die aard van *Delphi*

Programmering in ouer tale, soos *Pascal*, word vanuit ’n prosedurele benadering gedoen (Sebesta, 2012:41). *Delphi* staan as ’n hibriede programmeertaal bekend, aangesien dit die prosedurele benadering van die programmeertaal *Pascal*, waaruit dit ontwikkel het, behou het, met die byvoeging van komponente (objekte) vir ’n objekgeoriënteerde benadering (Sebesta, 2012:110). In *Delphi* kan programmering dus vanuit ’n prosedurele of vanuit ’n objekgeoriënteerde benadering geskied.

Die verskil tussen ’n prosedurele en objekgeoriënteerde benadering lê volgens Weisfeld (2009:6) in die wyse waarop data (attribute) en metodes (gedrag) in programmering hanteer word. In ’n prosedurele benadering word data en metodes

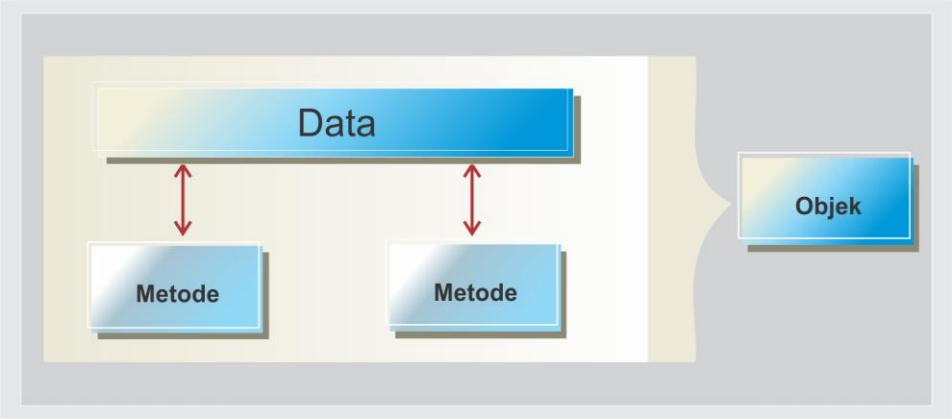
geskei (sien figuur 2.4) en in ’n objekgeoriënteerde benadering word data en metodes in ’n bepaalde objek bevat (sien figuur 2.5). Verskeie benamings word vir *metodes* gebruik na gelang van die benadering waaruit geprogrammeer word. Met ’n prosedurele benadering word na metodes as *subprogramme* (Sebesta, 2012:415) of *funksies* (Weisfeld, 2009:7) verwys. Die benaming metodes word algemeen in ’n objekgeoriënteerde benadering gebruik (Weisfeld, 2009:9). In hierdie studie gaan die benaming *subprogramme* gebruik word wanneer na ’n prosedurele benadering verwys word en *metodes* wanneer na ’n objekgeoriënteerde benadering verwys word. Vervolgens gaan die term subprogramme of metodes verder omskryf word.

Subprogramme (metodes) kan as ’n versameling instruksies wat saamgegroepeer word om bepaalde berekenings te doen, of om bewerkings in programme uit te voer, gedefinieer word (Sebesta, 2012:415). Daar bestaan twee soorte subprogramme, naamlik funksies of prosedures (Sebesta, 2012:408). Subprogramme word onder andere geskryf om leesbaarheid van programme te verhoog en programmeringstyd te verminder, aangesien programmeringskode hergebruik kan word indien dit in ’n subprogram geplaas is (Sebesta, 2012:408). In figuur 2.4 word aangetoon hoe data met ’n prosedurele benadering van subprogramme geskei kan word en globaal hanteer word, sodat ander dele van ’n program vrylik toegang tot die data het en ook daaraan kan verander (Weisfeld, 2009:7). Gevolglik kan ongekontroleerde toegang tot en verandering aan data plaasvind.



Figuur 2.4: Verskillende subprogramme het toegang tot data en kan daaraan verander (Weisfeld, 2009:7)

Indien programmering vanuit ’n objekgeoriënteerde benadering gedoen word, is die data en die prosedures deel van ’n objek. Figuur 2.5 dui aan hoe die data beskerm of enkapsuleer word in die objek, sodat geen toegang tot die data verkry kan word sonder dat die objek toegang daartoe verleen nie (Weisfeld, 2009:10). Objekte kan as die boublokke van ’n objekgeoriënteerde program gesien word en ’n objekgeoriënteerde program kan dus as ’n versameling objekte beskryf word (Weisfeld, 2009:10).



Figuur 2.5: Data en metodes word in ’n objek vervat (Weisfeld, 2009:10)

Volgens Sebesta (2012:545) kan ’n programmeertaal as ’n OOP klassifiseer word indien dit abstrakte datatipes, oorerflikheid en polimorfisme bevat. Kerman (2002:423) beweer dat die insluiting van enkapsulasie ook by bogenoemde benodig word vir klassifikasie as ’n OOP. Hierdie objekgeoriënteerde begrippe word vervolgens verduidelik.

’n Abstrakte datatipe, wat ook ’n *klas* genoem word, is ’n datastruktuur wat subprogramme bevat om data te manipuleer (Sebesta, 2012:495). ’n Instansie van hierdie abstrakte datatipe word ’n *objek* genoem. ’n Klas kan ook beskryf word as ’n bloudruk van ’n objek (Weisfeld, 2009:14). Net soos wat aan ’n persoon eienskappe (oogkleur, ouderdom en gewig) en gedrag (loop, praat en asemhaal) toegeken word, word ’n objek ook deur eienskappe en gedrag gedefinieer (Weisfeld, 2009:6). Die data van ’n objek verwys na die eienskappe daarvan, en die metodes wat die objek kan uitvoer om die data te manipuleer, word die *gedrag* van die objek genoem (Weisfeld, 2009:11). Enkapsulasie beteken dat die data van die objek versteek word

vir gebruikers van die objek en dat die data slegs verander kan word deur metodes wat in die objek ingeskryf is (Kerman, 2002:423) (sien figuur 2.5). ’n Nuwe, abstrakte datatipe kan die eienskappe en gedrag van ’n ander bestaande tipe erf en aan die entiteite daarvan verander – hierdie begrip staan as *oorerwing* bekend (Sebesta, 2012:545). *Polimorfisme* beteken dat ’n program objekte verskillend kan verwerk, na gelang van die behoefte van die klas (Kerman, 2002:423).

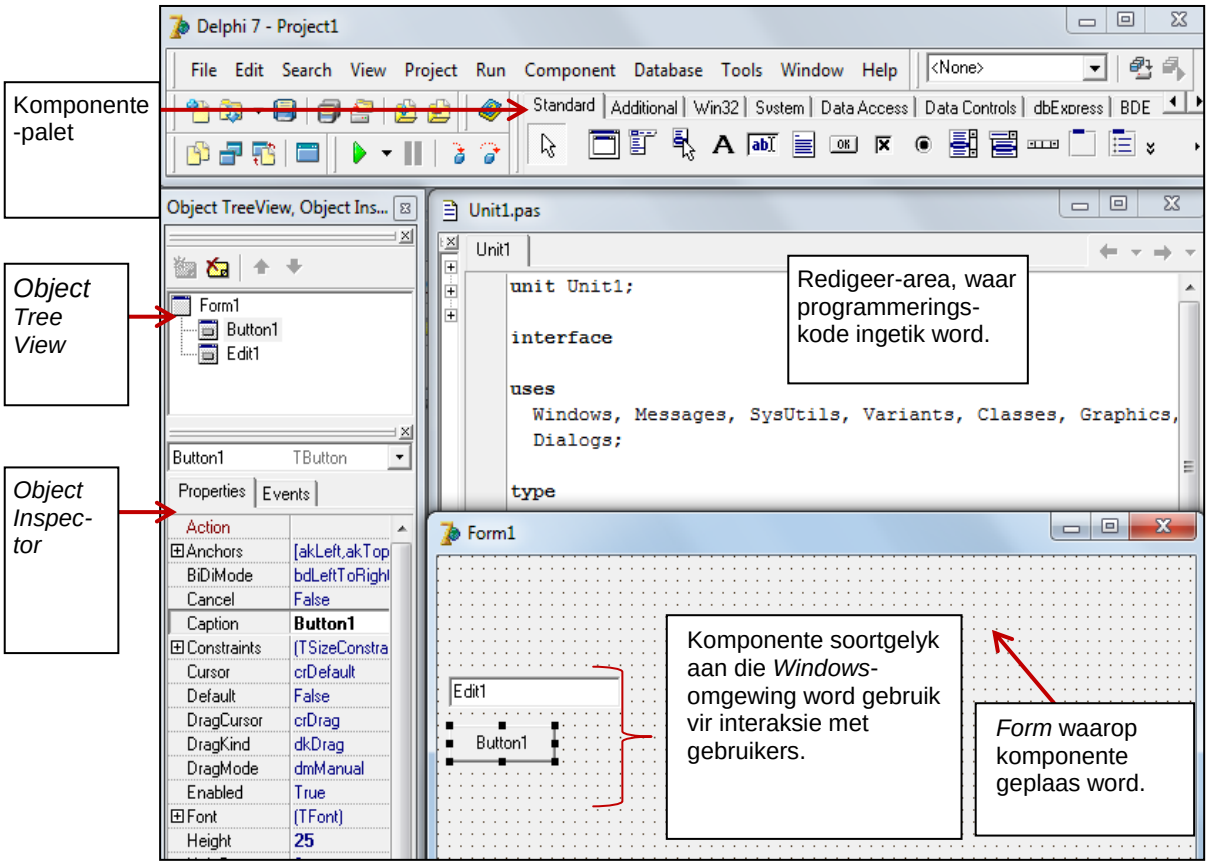
Aangesien al bogenoemde eienskappe in *Delphi* ingebou is, kan dit as ’n OOP geklassifiseer word (Kerman, 2002:424). In hierdie navorsing word na *Delphi* as ’n SOOP verwys, wat beteken dat dit ’n OOP is waar die programmeerder self die programmeringskode, volgens bepaalde sintaktiese reëls van die taal, moet intik. In die volgende onderafdelings word in meer besonderhede na die *Delphi*-programmeringsomgewing gekyk.

2.5.3 Die *Delphi*-programmeringsomgewing

’n *Delphi*-toepassing of -program kommunikeer met gebruikers deur middel van ’n grafiese koppelvlak (GK). Ook tydens die ontwikkeling van ’n toepassing word ’n GK gebruik (Sebesta, 2012:110). *Delphi* bevat al die nodige komponente (objekte) sodat programmerders funksionele GK'e, wat aan die behoeftes van gebruikers voldoen, kan ontwerp (Kerman, 2002:22). Heelwat tyd word gevolglik aan die ontwerp van koppelvlakke bestee. GK-komponente soortgelyk aan ’n *Windows*-omgewing (Kerman, 2002:22) word aan gebruikers verskaf om byvoorbeeld op te klik wanneer bepaalde aksies uitgevoer moet word of om data in te sleutel. ’n *Delphi*-program word gevolglik uitgevoer na gelang van hierdie aksies of gebeurtenisse wat plaasvind.

Die aanleer van die *Delphi*-GOO word deur Kerman (2002:xvi) as een van die struikelblokke vir nuwe programmeerders beskou, aangesien nuwe programmeerders eers daarmee vertroud moet wees voordat hulle kan begin met die aanleer van programmering in *Delphi*. In figuur 2.6 word die *Delphi*-GOO voorgestel met die onderskeie dele waaruit dit bestaan. Die GOO bestaan uit drie hoofdele. Die *Form* of vorm, is die deel waar die programmeerder die koppelvlak ontwerp en objekte vir interaksie met die gebruiker opsit. Die *Object Tree View* verskaf ’n hiërargiese uitleg van alle objekte wat op die vorm geplaas word. Die *Object Inspector* bevat inligting oor die eienskappe van objekte, sowel as gebeurtenisse wat

aan elke objek gekoppel is. Bo-aan die GOO is ’n kieslys met fasiliteite soos byvoorbeeld stoor en uitvoer van programme, en ’n komponente-palet waaruit komponente of objekte gekies kan word om op die vorm te plaas (Kerman, 2002:23–34).



Figuur 2.6: *Delphi*-geïntegreerde ontwikkelingsomgewing (GOO)

Delphi word met die term vinnige toepassingsontwikkelling (VTO), of *Rapid Application Development*, geassosieer (Kerman, 2002:22). Toepassings kan gevolglik relatief vinnig ontwikkel word, omdat programmeerders uit die komponente-palet objekte kan kies om op die vorm te plaas. Soos wat die objekte op die vorm geplaas word, word ’n voorlopige raamwerk van programmeringskode aan programmeerders verskaf. ’n Voorbeeld van ’n voorlopige raamwerk van ’n *Delphi*-program, met ’n *Button*-objek wat op die vorm geplaas is, word in figuur 2.7 gegee.

’n *Delphi*-toepassing of projek is ’n samestelling van verskeie lêers, naamlik projek-lêers, eenheid-lêers en vorm-lêers (Kerman, 2002:48). Die projek-lêer bevat

instruksies om al die onderskeie eenhede (*Units*) *(*sien figuur 2.7) en vorms waaruit die projek bestaan, aanmekaar te koppel. Die voorafgaande impliseer dat die stoor van selfs 'n eenvoudige *Delphi*-projek 'n redelike komplekse proses is, aangesien verskeie lêers in die proses gestoor moet word.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin

end;

end.
```

Figuur 2.7: Raamwerk van *Delphi*-programmeringskode

2.5.4 *Delphi* as programmeertaal

Uit die voorafgaande besprekings oor die aard en doel van *Delphi* behoort dit duidelik te wees dat *Delphi* 'n volwaardige programmeertaal is waarmee komplekse toepassings ontwikkel kan word.

2.5.4.1 Die taal van Delphi

Delphi staan volgens Kerman (2002:28) as 'n objekgeoriënteerde programmeertaal bekend. Alle komponente in *Delphi* word objekte genoem. Hierdie objekte het

verskeie eienskappe, en bepaalde gebeurtenisse kan daaraan gekoppel word (Kerman, 2002:28). In sommige opsigte lyk *Delphi*-taal soos Engels, maar woorde word volgens *Delphi*-reëls gespel. *Delphi* is dus ’n programmeertaal (Kerman 2002:128) wat net soos ’n spreektaal, uit simbole (’n alfabet), woorde, sintaksis en semantiek bestaan. Programmeringskode moet noukeurig ingesleutel word en die sintaktiese reëls van die taal moet tydens programmering nagekom word (Sajaniemi & Kuittinen, 2008:77). Figuur 2.8 toon ’n voorbeeld van *Delphi*-programmeringskode aan, wat die woord "Hallo" en die naam wat die gebruiker ingevoer het, op die skerm vertoon.

```
procedure TForm1.btnGroetClick(Sender: TObject);

begin

  pnlAfvoer.Caption := 'Hallo ' + edtNaam.Text;

end;
```

Figuur 2.8: Voorbeeld van *Delphi*-programmeringskode

2.5.4.2 Sintaksis

Die sintaktiese reëls van ’n taal bepaal watter stringe (kombinasie van karakters) uit die alfabet in die taal mag voorkom (Sebesta, 2012:135). *Delphi*-programmeringskode moet deur ’n vertalerprogram na masjienkode vertaal word om uitvoerbare kode te skep wat uiteindelik programuitvoering moontlik maak (Kerman, 2002:51). Indien die instruksies nie presies volgens die sintaktiese reëls van die programmeertaal ingetik word nie, kan die vertaler nie die instruksies na masjienkode vertaal nie en kan programuitvoering nie plaasvind nie (Sebesta, 2012:45). In ’n programmeertaal soos *Delphi* moet elke instruksie korrek gespel word en elke karakter moet op die regte plek getik word, om foutboodskappe betreffende sintaksis tydens vertaling te voorkom (Kerman, 2002:86). Indien ’n sintaksisfout begaan word, word dit deur middel van ’n rooi-gemerkte boodskap deur die vertaler aangedui en die vertalingsproses word gestaak. Die boodskappe is nie altyd gebruikersvriendelik nie

en dit neem ’n rukkie vir programmeerders om te verstaan wat die boodskappe beteken en hoe die foute reggemaak kan word (Gomes & Mendes, 2007:2).

2.5.4.3 Veranderlikes en datatipes

Delphi bevat duidelike sintaktiese reëls oor die benoeming van veranderlikes. Die eerste letter van ’n veranderlike se naam moet byvoorbeeld ’n alfabetiese letter wees en die naam mag nie ’n spasie insluit nie. ’n Groot verskeidenheid datatipes bestaan en duidelike beperkings word op die maksimum- en minimumwaardes daarvan geplaas. Tydens die skryf van *Delphi*-programme moet programmeerders dus goeie kennis van die benoeming van veranderlikes se name en die gebruik van die onderskeie datatipes hê, heeltyd daarop let en gedurige omskakelings tussen datatipes doen. Sekere toevoerkomponente kan byvoorbeeld net string-datatipes hanteer, wat beteken dat, indien ’n getal ingesleutel moet word, die stringwaarde omgeskakel moet word na ’n waarde wat ’n getal-datatipe is. Die omgekeerde moet weer tydens afvoer gebeur, omdat getal-datatipes weer in ’n netjiese formaat na string-datatipes omgeskakel moet word om dit in afvoerkomponente te plaas (Kerman, 2002:77, 78, 93,102).

’n Verdere aspek in *Delphi* wat veranderlikes betref, is die reikwydte van veranderlikes. Dit bepaal waar in ’n program veranderlikes gebruik kan word. Die plek waar ’n veranderlike in ’n program verklaar word, bepaal die reikwydte daarvan. Die begrip kan ook wyer op objekte toegepas word, aangesien die reikwydte van objekte ook op die beskikbaarheid of sigbaarheid daarvan dui wat betref die verskillende dele van die program of ander programme (Kerman, 2002:94). Kerman (2002:97) beveel aan dat goeie programmeringstegnieke oor die gebruik van reikwydte in inleidende programmeringskursusse aangeleer moet word.

Die begrip van ’n veranderlike is ’n abstrakte gedagte wat oor die algemeen vir beginnerprogrammeerders moeilik is om te begryp (Lahtinen *et al.,* 2005:16). Voeg nog al die sintaktiese reëls, datatipes, beperkings, omskakeling van tipes en reikwydte by, soos bo beskryf, en dit behoort duidelik te wees dat die hantering van veranderlikes in *Delphi*-programmering op sy eie vir beginnerprogrammeerders ’n komplekse saak is.

2.5.4.4 Herhaling- en besluitnemingstrukture

Delphi het verskeie herhalingstrukture, naamlik FOR, WHILE en REPEAT. Hierdie herhalingstrukture kan as onvoorwaardelike herhaling (voer ’n sekere aantal kere uit) of voorwaardelike herhaling (herhaal totdat ’n sekere voorwaarde bevredig word) geklassifiseer word. Elke herhalingstruktuur het ’n bepaalde funksionaliteit, sintaksis en reëls vir die programmering daarvan (Kerman, 2002:150,156–165). ’n Programmeerder moet elke herhalingstruktuur verstaan om te kan besluit watter struktuur die doeltreffendste oplossing vir ’n probleem sal bied (Sebesta, 2012:369). Elke struktuur bevat ’n kombinasie van instruksies wat op ’n bepaalde manier opgestel moet word om ’n oplossing te implementeer (Rogalski & Samurcay, 2010:9). Kennis en begrip van die probleem wat opgelos moet word, sowel as kennis en begrip van die funksie, sintaksis en struktuur van voorwaardelike en onvoorwaardelike herhalingstrukture (Baldwin & Kuljis, 2000:285) is dus noodsaaklik vir suksesvolle gebruik in ’n *Delphi*-program.

Net soos herhalingstrukture, is daar verskeie besluitnemingstrukture in *Delphi*, naamlik IF-THEN-ELSE, IF-ELSE IF en CASE (Kerman, 2002:128). Soortgelyke strukture word in die meeste ander programmeertale gevind (Sebesta, 2012:370–382). Hierdie besluitnemingstrukture funksioneer op Boole-voorwaardes wat deur die programmeerder opgestel moet word (Kerman, 2002:122–127). Boole-voorwaardes kan kompleks wees en kennis van simbole, operators en sintaksis, asook logiese denke oor die samestelling daarvan, word benodig (Sebesta, 2012:370–377). ’n Boole-voorwaarde word in ’n besluitnemingstruktuur geïnkorporeer om besluite te neem en uitvoering daarvolgens aan te pas. Elke besluitnemingstruktuur het vaste sintaktiese reëls. Die samestellings van strukture kan verskil na gelang van die probleem wat opgelos moet word, en programmeerders moet in staat wees om op besluitnemingsmodelle te bou en dit vir spesifieke probleemscenario’s aan te pas (Sebesta, 2012:370).

2.5.5 Programmering in *Delphi*

Programmering in *Delphi* behels al vyf domeine van Du Boulay (1989), soos in paragraaf 2.3 bespreek, naamlik probleemoplossing, die ontwerp van algoritmes, begrip van hoe die *Delphi*-omgewing werk, die ontwerp van ’n koppelvlak vir

interaksie tussen die gebruiker en die program, die intik van programmeringskode en toetsing van programme. Net soos die bedryfstelsel *Windows* word ’n *Delphi*-program ook deur gebeurtenisse gedryf (Kerman, 2002:97) (sien 2.5.3). Dit beteken dat programinstruksies gekoppel word aan gebeurtenisse wat plaasvind, soos byvoorbeeld die oopmaak van ’n vorm en die klik op ’n sekere knoppie. Programmering in *Delphi* word vervolgens met betrekking tot die proses van programmering bespreek.

2.5.5.1 Probleemoplossing en ontwerp

Net soos in ander programmeertale (sien 2.3) is probleemoplossing en die ontwerp van algoritmes (Gibson & O’Kelly, 2005:87) van die belangrikste stappe in *Delphi*-programmering (Kerman, 2002:63). Die tik van programmeringskode kan eintlik net as ’n vertaling van die algoritme in *Delphi*-taal beskou word (Jenkins, 2002:55). Tydens die vertaling en uitvoering van programmeringskode moet probleemoplossing en foutopsporing ook gedurig toegepas word, aangesien foute tydens vertaling en uitvoering van programme kan voorkom (sien 2.3) en gevolglik opgelos moet word.

Die ontwerp van ’n gebruikerskoppelvlak is deel van die probleemoplossingsproses. Daar is verskeie objekte wat op die vorm geplaas kan word vir toevoer van data en afvoer van resultate (sien 2.5.3) en die programmeerder moet die beste objek vir die spesifieke toepassing kan kies (Kerman, 2002:45). Aangesien *Delphi* deur gebeurtenisse gedryf word, moet daar besluit word watter gebeurtenisse aktiveer gaan word en programmeringskode gaan uitvoer. Dit moet in die ontwerp van die koppelvlak geïntegreer word (Kerman, 2002:45). Die programmeerder moet dus ’n geheelbeeld van die finale oplossing kan vorm voordat die programmeringskode ingesleutel word, omdat die koppelvlak aan die begin van die koderingsproses as deel van die implementering van die algoritme ontwerp word.

Die menslike faktor is ’n verdere aspek wat in probleemoplossing die hoof gebied moet word. Gebruikers kan ongeldige en foutiewe waardes insleutel en dit kan veroorsaak dat foutiewe resultate deur ’n program afgevoer word, of selfs dat programuitvoering staak (Kerman, 2002:222). Instruksies en programmeringstrukture moet dus sodanige uitvoerfoute kan voorkom.

2.5.5.2 Implementering

Die implementering van ’n *Delphi*-oplossing is ’n komplekse proses. Die *Delphi*-GOO moet eerstens opgestel word, programmeringskode moet vir die spesifieke gebeurtenisse geskryf word en die onderskeie lêers moet op toepaslike wyse gestoor word (sien 2.5.3). Die programmeerder moet deurgaans sintaksisfoute hanteer en veral op logiese foute let (sien 2.5.5.3).

2.5.5.3 Instandhouding, fouthantering en toetsing

’n *Delphi*-programmeerder is net soos enige ander programmeerder deurentyd besig met fouthantering. Daar word hoofsaaklik drie soorte foute in rekenaarprogramme aangetref, naamlik sintaksisfoute, logiese foute en uitvoerfoute (Kerman, 2002:222). Dit is menslik vir ’n programmeerder om ’n *sintaksisfout* te maak en ’n aanhalingsteken te vergeet of ’n hakie op ’n verkeerde plek te plaas. Vir die vertalerprogram is elke klein foutjie egter ’n kritieke saak, want dit beteken dat die vertaling van die programmeringskode na masjienkode nie kan plaasvind nie (sien 2.5.4.2). Ontfouting verg uithouvermoë (Gomes & Mendes, 2007:4), aangesien foute nie altyd voor die hand liggend is en vinnig en maklik opgespoor word nie.

’n Foutiewe algoritme of ’n foutiewe waarde wat ’n gebruiker ingetik het, kan veroorsaak dat, alhoewel ’n program uitvoer en selfs afvoer lewer, die afvoer nie korrek is nie. Hierdie soort foute, naamlik *logiese foute*, word met die semantiek van die taal geassosieer (Kerman, 2002:223). Programme wat ‘klaar’ geskryf is en nie meer sintaksisfoute bevat nie, moet gevolglik getoets word om te bepaal of dit die probleem korrek oplos. Dit word gedoen deur verwagte en ekstreme waardes as toevoer in te tik sodat daar so ver moontlik bepaal kan word of ’n program onder alle omstandighede korrekte afvoer sal lewer. Alhoewel die *Delphi*-GOO help met die opspoor van sintaksisfoute, bestaan daar nie altyd maklike maniere om dit op te spoor nie. Om logiese foute op te spoor, moet programmeerders vertroud wees met alternatiewe foutopsporingstegnieke. ’n Program kan stap-vir-stap per hand uitgevoer word om te kyk waar die probleem ontstaan het en hoe dit reggemaak kan word. *Delphi* het ook ’n ingeboude ontfoutingsfasiliteit wat die programmeerder toelaat om ’n bepaalde deel van die program stap-vir-stap uit te voer en die waardes van

veranderlikes tydens uitvoering dop te hou (Kerman, 2002:234–237). Albei hierdie tegnieke is egter tydsame prosesse wat geduld en deursettingsvermoë verg.

Uitvoerfoute is die laaste kategorie foute wat bespreek word. Tydens *uitvoerfoute* is daar geen sintaksis- of logiese foute in die program nie, maar ’n ander, eksterne probleem wat veroorsaak dat die program staak (Kerman, 2002:229). Dit impliseer dat programmeerders aan alle moontlike scenario’s wat verkeerd kan gaan tydens uitvoering van programme moet dink en deur middel van programmeringskode daarvoor voorsiening moet maak. Voorbeelde van sodanige situasies is dat die gebruiker ’n datum in die verkeerde formaat ingetik het, of dat die program ’n lêer probeer oopmaak het wat nie bestaan nie. Heelwat programmeringskode moet gewoonlik ingetik word om vir hierdie soort foute voorsiening te maak – soms meer as wat vir die oplossing van die probleem benodig word (Kerman, 2002:229,231).

2.6 VERGELYKING TUSSEN *SCRATCH* EN *DELPHI* AS PROGRAMMEERTALE

In hierdie afdeling word *Scratch* en *Delphi* aan die hand van die voorafgaande bespreking met mekaar vergelyk (sien tabel 2.4) met die oog op die laaste doelwit, naamlik om te bepaal watter programmeringsbegrippe reeds met die onderrig van *Scratch* vasgelê kan word.

Uit die vergelykings in tabel 2.4, kan die gevolgtrekking gemaak word dat daar, wat elke programmeringsbegrip aanbetref, ooreenkomste tussen die twee programmeertale bestaan. Daar is wel leemtes in *Scratch* wat bepaalde programmeringsbegrippe aanbetref, soos byvoorbeeld dat datatipes nie aan veranderlikes toegeken word nie en dat *Scratch* nie ’n volwaardige objekgeoriënteerde taal is nie, maar verwysings kan steeds na datatipes en objekgeoriënteerde begrippe in *Scratch* gemaak word. In beide tale moet probleemoplossing plaasvind, ontwikkeling in ’n GOO geskied en programme geïmplementeer en uitgevoer word. Daar bestaan sterk ooreenkomste tussen herhaling- en besluitnemingstrukture, en fouthantering en toetsing is ’n integrale deel van programmering in beide tale.

Tabel 2.4: Vergelyking tussen *Scratch* en *Delphi*

BESKRYWING	*SCRATCH*	*DELPHI*
1. Probleemoplossing	• Lees en verstaan probleem. • Ontwerp algoritme om probleem op te los. • Stel *Sprite*-area en *Script*s op. • Toets uitvoering van program. • Probleemoplossing kan ook tydens programuitvoering gedoen word – verander blokke kode terwyl program uitvoer om uitwerking daarvan te sien. (sien 2.4.5.1)	• Lees en verstaan probleem. • Ontwerp algoritme om probleem op te los. • Stel koppelvlak op en tik programmeringskode in. • Ontfout kode en toets uitvoering van program. • Effek van programuitvoering kan slegs gesien word wanneer al die programmeringskode ontfout is. • Voorafontwerp van oplossing moet deeglik gedoen word. (sien 2.5.5.1)
2. Geïntegreerde ontwikkelings-omgewing	Grafiese koppelvlak wat bestaan uit 'n kodeblok-area, *Sprites* en 'n *Sprite*-area. (sien 2.4.3)	• Grafiese koppelvlak met GUI-komponente, soortgelyk aan 'n *Windows*-omgewing. • Bestaan uit 'n *Object Tree View*, *Form* en *Object Inspector*. (sien 2.5.3)

BESKRYWING	*SCRATCH*	*DELPHI*
3. Implementering van program	• Plaas *Sprites* in kodeblok-area. • Koppel *Script*s aan *Sprites* deur die insleep van blokke kode. (sien 2.4.4.2)	• Plaas objekte vir toevoer, afvoer en verwerking op 'n vorm deur op kieslys te klik. • Koppel gebeurtenisse aan komponente (objekte) deur kode in te tik. (sien 2.5.4.1)
4. Sintaksis	Ingebou in kodeblokke. (sien 2.4.4.3)	Deeglike kennis van sintaksis word benodig om programmeringskode van nuuts af in te tik. (sien 2.5.4.2)
5. Veranderlikes en datatipes	Enige benoeming kan vir veranderlikes gegee word.	Noodsaak bepaalde sintaktiese reëls oor die benoeming van veranderlikes.
	Monitors verskaf 'n visuele voorstelling van die inhoud van veranderlikes.	Veranderlikes se waardes is nie sigbaar tydens uitvoering van program nie.
	• Parameter-gleuwe se vorms is 'n aanduiding van datatipes, naamlik Boole, getalle en stringe. • Onderskeid word nie tussen datatipes gemaak nie en omskakeling van datatipes word nie benodig nie. (sien 2.4.4.4)	• 'n Verskeidenheid datatipes, wat elk op 'n bepaalde manier hanteer moet word. Elke datatipe het 'n begrensing van moontlike waardes. • Omskakeling van datatipes moet tydens toevoer en afvoer plaasvind. (sien 2.5.4.3)

BESKRYWING	*SCRATCH*	*DELPHI*
Veranderlikes en datatipes (vervolg)	Reikwydte van veranderlikes hang af of ’n veranderlike aan ’n *Sprite* of ’n *Stage* behoort.	Reikwydte van veranderlikes hang af van waar ’n veranderlike verklaar word.
6. Programuitvoering	• Eenvoudige aksie, klik op groen vlaggie. • Programme voer gewoonlik uit. (sien 2.4.4.6)	• Aktiveer vertaler om program na masjienkode te vertaal. • Program word na uitvoerbare kode vertaal indien alle sintaksisfoute reggemaak is. (sien 2.5.4.2)
7. Herhaling- en besluitnemingstrukture	• Visueel voorgestel en vooraf in blokke ingebou. • Verskeidenheid strukture om van te kies.	Deeglike kennis van herhaling en besluitmening word benodig om effektiewe programme te skryf.
	Begrip van voorwaardelike en onvoorwaardelike herhaling en besluitneming en die toepassing daarvan word benodig.	Begrip van voorwaardelike en onvoorwaardelike herhaling en besluitneming en die toepassing daarvan word benodig.
	Werking van onderskeie herhaling- en besluitnemingstrukture moet begryp word om effektiewe programme te skryf.	• Deeglike kennis van sintaktiese reëls van strukture benodig. • Model van strukture moet vasgelê wees omdat die opstelling daarvan kan varieer volgens die probleem wat opgelos moet word.

BESKRYWING		*SCRATCH*	*DELPHI*
	Herhaling- en besluitnemingstrukture (vervolg)	Deeglike begrip van Boole-voorwaardes en opstelling daarvan. (sien 2.4.4.5)	Deeglike begrip van Boole-voorwaardes en opstelling daarvan. (sien 2.5.4.4)
8.	Objekgeoriënteerde benadering.	• Nie 'n ware objekgeoriënteerde taal nie, maar analogieë na begrippe van objekoriëntering kan op kreatiewe maniere gemaak word. • *Sprite* kan as 'n objek met eienskappe en gedrag beskou word.	Bevat alle eienskappe van 'n objekgeoriënteerde taal, naamlik abstraksie, polimorfisme, oorerwing en enkapsulasie.
		• Hoofsaaklik prosedurele benadering binne een *Script*. • Elke *Sprite* het sy eie *Script* wat aan hom gekoppel is en *Script*s kan gelyktydig uitvoer. (sien 2.4.4.1)	Prosedurele of objekgeoriënteerde benadering kan gevolg word. (sien 2.7.2)
9.	Fouthantering en toetsing	Logiese foute en uitvoerfoute kan tydens programuitvoering plaasvind.	Sintaksisfoute, logiese foute en uitvoerfoute kan tydens programuitvoering plaasvind.
		Geen sintaksisfoute kan voorkom nie, aangesien kodeblokke ingesleep word.	Sintaksisfoute in programmeringskode moet eers reggemaak word voordat program kan uitvoer.
		Vaardighede word benodig om te toets of 'n program 'n bepaalde probleem reg oplos.	Vaardighede benodig om te toets of 'n program 'n bepaalde probleem reg oplos.

BESKRYWING	*SCRATCH*	*DELPHI*
Fouthantering en toetsing (vervolg)	Moet programme deur middel van toetsdata kan toets.	Moet programme deur middel van toetsdata kan toets.
	• Monitors vertoon inhoud van veranderlikes tydens uitvoering. • Uitvoering van programme kan stapsgewys, teen stadige tempo, gedoen word om logiese foute op te spoor.	Het ’n ingeboude ontfoutingsfasiliteit wat die programmeerder toelaat om ’n bepaalde deel van die program stap-vir-stap uit te voer en die waardes van veranderlikes tydens uitvoering dop te hou
	Gereedskap benodig om logiese foute op te spoor, soos byvoorbeeld naspeurtabelle	Gereedskap benodig om logiese foute op te spoor, soos byvoorbeeld naspeurtabelle en programontfouters.
	Bewustheid moet gekweek word vir situasies waar uitvoering kan staak, soos wanneer deling deur nul plaasvind. (sien 2.4.5.3)	Bewustheid moet gekweek word vir situasies wat uitvoering kan staak, soos wanneer deling deur nul plaasvind.
		Programmering moet voorsiening maak vir uitvoerfoute en invoer van ongeldige waardes. (sien 2.5.5.2)

2.7 DIE ONDERRIG–LEER VAN PROGRAMMERING

In die voorafgaande paragrawe is die aard van programmering bespreek en is daar gesien dat raakvlakke tussen *Scratch* en *Delphi* wel geïdentifiseer kan word. Vervolgens word die onderrig–leer daarvan ondersoek. Die bespreking word gebaseer op die aard van programmering, soos in 2.3 bespreek, watter kennis daarvoor benodig word, sowel as moontlike onderrig–leerstrategieë wat gebruik kan word tydens die onderrig–leer van programmering. Die ondersoek fokus op die onderrig–leer van *Scratch*, aangesien die uiteindelike doel is om te bepaal hoe *Scratch* onderrig kan word met die oog op die oorgang na *Delphi*.

Donchev en Todorova (2013:84) voer aan dat die onderrig van programmering ná 50 jaar steeds ’n uitdaging is, veral wat die onderrig van beginnerprogrammeerders betref, ten spyte van baie navorsing wat al op die gebied gedoen is. Programmeerders moet weet hoe om gestruktureerde oplossings vir probleme te ontwerp, hoe om instruksies aan ’n rekenaar te gee om ’n oplossing te implementeer en te toets, wat ’n programmeertaal se rigiede komplekse sintaksis behels, hoe om instruksies te organiseer en hoe ’n rekenaar die instruksies sal uitvoer (Donchev & Todorova, 2013:84; Gomes & Mendes, 2007; Kelleher & Pausch, 2005:86). Gomes en Mendes (2007:1) voeg vaardighede soos veralgemening, oordrag en kritiese denke by bogenoemde. Volgens Resnick (2012) behels die onderrig van programmering ook die onderrig van die proses om ’n program te ontwerp, sowel as die beginsels van ontwerp. Die ontwerp van ’n program behels om met nuwe idees te eksperimenteer, om ’n idee tot ’n volledige program te ontwikkel, en om komplekse idees in kleiner, eenvoudige oplossings te verdeel. Resnick (2012) noem ook dat leerders moet leer om met ander leerders saam te werk aan die oplossing van ’n probleem. Hulle moet foute in programme kan opspoor, dit regmaak, en leer om aan te hou en uit te hou as ’n program nie reg wil werk nie. Jenkins (2002:55) sluit by Resnick (2012) aan, en voer aan dat leerders ’n hiërargie van vaardighede vir programmering moet aanleer, waarvan kodering (die skryf van programmeringskode) slegs ’n klein deeltjie behels.

Jenkins (2002:55) identifiseer verskeie prosesse in programmering, onder andere die ontwerp van algoritmes, en die omskakeling van algoritmes na programmeringskode, waarvan die ontwerp van algoritmes as die moeilikste en belangrikste gereken word.

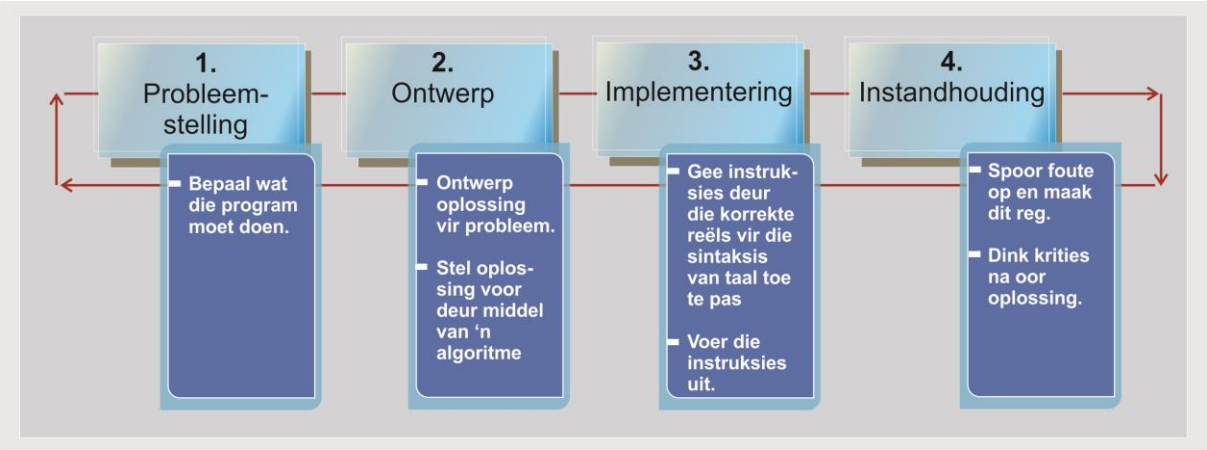
Reeds in die aanvangsfase van programmering moet leerders algoritmes kan skryf wat probleme oplos. Volgens Gomes en Mendes (2007:1) begin baie leerders se probleme met programmering reeds hier, met die skryf van algoritmes, en moet baie aandag tydens die onderrigproses daaraan gegee word. Yadin (2013:16) meld egter dat die onderrig van die sintaksis en semantiek van ’n programmeertaal in baie gevalle die eerste paar weke van programmeringsonderrig oorheers.

Volgens Kelleher en Pausch (2005:86) sukkel beginnerprogrammeerders meestal om dit wat hulle wil hê programme moet doen, in sintakties korrekte instruksies te stel sodat die rekenaar dit kan verstaan. Hulle voer aan dat een manier om die probleem te oorkom, onderrig in ’n programmeertaal soos *Scratch* is, wat sintakties maklik is, maar wat steeds die fundamentele strukture van ’n SOOP bevat. Sodoende kan onderrig op programmeringsbeginsels en -begrippe fokus. Leerders behoort dan volgens Kelleher en Pausch (2005:87–88) die kennis wat hulle met die onderrigtaal opgedoen het, na ’n ander programmeertaal te kan oordra. Hierdie bewering van Kelleher en Pausch bevestig die feit dat daar met goeie beplanning wel programmeringsbeginsels en -begrippe in *Scratch* gevestig kan word, wat waardevol in *Delphi*-programmering kan wees.

’n Verdere groot struikelblok vir beginnerprogrammeerders is sintaksis, gevolglik maak hulle sintaksisfoute (Kelleher & Pausch, 2005:89). Al is ’n program sintakties reg, is dit steeds moeilik vir beginnerprogrammeerders om te verstaan hoe ’n program uitgevoer word en hoe om foute op te spoor. Kelleher en Pausch (2005:102) sê dat maniere gevind moet word om die uitvoering van programme na te spoor ten einde abstrakte begrippe, soos veranderlikes en objekte, meer konkreet te maak.

Uit bostaande behoort dit duidelik te wees dat die onderrig van programmering veel meer as net die aanleer van ’n programmeertaal (Sajaniemi & Kuittinen, 2008:76) en die onderrig van programmeringsbeginsels behels. Al vier fases in die proses van programmering, soos in figuur 2.1 aangedui, moet tydens die onderrig van programmering die hoof gebied word.

Na aanleiding van die bespreking in die vorige paragrawe oor die onderrig van programmering, kan Rogalski en Samurcay (2010:8) se proses van programmering (sien figuur 2.1) verder aangevul word, soos in figuur 2.9 aangedui.



Figuur 2.9: Vaardighede wat tydens programmering onderrig moet word

Rogalski en Samurcay (2010:8) beweer dat die voorstelling van 'n probleem en die oplossing daarvoor 'n minder komplekse taak in die onderrig–leersituasie sal wees indien die probleem duidelik gedefinieer word en aan 'n bekende domein behoort, soos byvoorbeeld om die gemiddelde prestasie van leerders te bepaal. Volgens Rogalski en Samurcay kan die onderrig van programmering nie as 'n lineêre opbou van kennis gesien word nie, aangesien daar baie aspekte in die proses van programmering is wat onderrig moet word en verskillende soorte kennis vir die onderskeie aspekte benodig word. Soos in figuur 2.9 aangedui, word kennis benodig om oplossings vir probleme deur middel van algoritmes voor te stel. In die implementeringsfase word kennis benodig om algoritmes om te sit in programmeringskode volgens die korrekte sintaksis van die taal en moet die instruksies uitgevoer word. Tydens die instandhoudingsfase word kennis benodig om foute op te spoor en reg te maak en moet daar oor die oplossing gereflekteer word.

Uit die voorafgaande bespreking blyk dit dat die vaardighede of kognitiewe take wat in programmering onderrig moet word, in twee hoofgroepe geplaas kan word. Die eerste groep kan as die laer kognitiewe take of deklaratiewe kennis (Yadin, 2013:14,15) beskou word, naamlik memorisering van die sintaksis en semantiek van die programmeertaal en die omgaan met die GOO. Tradisioneel sal leerders tydens programmering eerstens onderrig word in die laer kognitiewe take, soos byvoorbeeld die sintaksis van die taal, datatipes en bewerkingsoperators (Gao, 2011:1267) en

daarna sal die onderrig opwaarts in die hiërargie beweeg na moeiliker begrippe soos veranderlikes en funksies (Donchev & Todorova, 2013:85).

Volgens Dijkstra (1989:1390) is leer egter ’n stadige en geleidelike proses en neem dit tyd om nuwe begrippe na bekende begrippe toe om te skakel. Hy beweer dat tradisionele, lesinggebaseerde onderrig–leerstrategieë nie voldoende is vir die onderrig van programmering nie. Dijkstra (1989) beskou programmering as intensiewe probleemoplossing wat ’n intense vlak van presisie behels, aangesien een kommapunt ’n verskil kan maak tussen sukses en mislukking. Dit is sulke sintaktiese kwessies, tesame met die worsteling om ’n oplossing vir ’n probleem te vind, wat daartoe bydra dat leerders soms eerder uithouvermoë as leervaardighede by die aanleer van programmering benodig (Gomes & Mendes, 2007:4). Kelleher en Pausch (2005:107) sê in dié verband ook dat motivering ’n sleutelelement tydens leer is en gevolglik behoort leerders tydens onderrig ondersteun te word om ’n bepaalde doel te bereik en struikelblokke te oorkom.

Vervolgens word die tweede groep, die hoër kognitiewe take of prosedurele kennis (Yadin, 2013:15), naamlik probleemoplossing, algoritme-ontwerp, logiese denke, foutopsporing en toetsing in oënskou geneem. Gomes en Mendes (2007:2) voer aan dat een van die groot redes waarom beginnerprogrammeerders nie kan programmeer nie, is omdat hulle nie oor probleemoplossingsvaardighede beskik nie, en gevolglik nie algoritmes kan ontwerp nie. Rogalski en Samurcay (2010:12) waarsku egter dat daar altyd in gedagte gehou moet word dat probleemoplossingsvaardighede langtermynprosesse is, wat ervare programmeerders oor baie jare heen ontwikkel het. Volgens Gomes en Mendes (2007:2) behels die oplos van probleme verskeie vaardighede. Hierdie vaardighede word vervolgens bespreek.

Verstaan die probleem

Leerders probeer om probleme op te los sonder dat hulle die probleme verstaan. Hulle is baie gretig om programmeringskode te skryf, maar bestee nie genoeg tyd aan die lees en interpretasie van die probleem nie (Gomes & Mendes, 2007:2). Daar word voorgestel dat leerders gehelp moet word met hulle begrip van probleme deur

byvoorbeeld aan hulle verskeie toevoerwaardes vir ’n bepaalde probleem te gee, sodat hulle kan bepaal wat die afvoer daarvan sal wees (Gomes & Mendes, 2007:3).

Verwante kennis

Gomes en Mendes (2007:2–3) voer aan dat leerders nie altyd analogieë met vorige probleme vorm en vorige kennis na nuwe probleme oordra nie. Leerders groepeer probleme nie volgens die programmeringsbeginsels en -begrippe wat daarin bevat is nie en baseer hulle oplossings op vorige probleme wat nie met die huidige oplossing verband hou nie. Gomes en Mendes (2007:2–3) stel dus voor dat probleme wat aan leerders gegee word, by vorige probleme moet aansluit en in moeilikheidsgraad moet toeneem.

Refleksie oor die probleem en die oplossing

Hier word verwys na metakognitiewe vaardighede soos selfregulering, denke oor die effektiwiteit van algoritmes en refleksie oor oplossings. Gomes en Mendes (2007:2) beweer dat leerders oplossings skryf sonder om daaroor na te dink. Wanneer die program volgens hulle werk, voel leerders tevrede, sonder om dit aan verskillende toetsdata te onderwerp. Om leerders se hoër kognitiewe en metakognitiewe vaardighede te ontwikkel, moet sodanige vaardighede pertinent onderrig word (Deek, 1999:4; Mevarech & Kramarski, 1997:368; Smitherman & Girard, 2011:53). Leerders moet geleer word om tydens probleemoplossing met gereelde tussenposes bepaalde vrae te vra en op hulle werk te reflekteer (Gomes & Mendes, 2007:4). Voorbeelde van sodanige vrae is: “Wat doen ek nou?” “Hoekom doen ek dit?” en “Hoe help dit my?” (Mevarech & Kramarski, 1997:369; Teasley, 2010:77). Aanvanklik moet leerders gemonitor en daaraan herinner word om hierdie vaardighede toe te pas, maar later vind dit natuurlik plaas wanneer hulle probleemoplossing doen (Breed, 2010:243).

Metakognitiewe vrae kan oor die algemeen in drie groepe ingedeel word, naamlik begripsvrae, strategiese vrae en konneksievrae (Mevarech & Kramarski, 1997:369–370). *Begripsvrae* help leerders om die probleem beter te verstaan (Sumiga, 2010:67). Leerders kan byvoorbeeld gevra word om die probleem in hulle eie woorde te beskryf, of om ’n diagram te teken wat die probleem voorstel. *Strategiese vrae* verwys na moontlike strategieë wat gebruik kan word om ’n probleem op te los, soos

byvoorbeeld bo-na-onder, onder-na-bo (Détienne, 2010:49), lukraak, of ’n hibriede kombinasie van die eerste twee strategieë. Leerders moet ’n strategie voorstel om ’n probleem op te los en verduidelik hoekom hulle so besluit het. *Konneksievrae* handel oor ooreenkomste en verskille tussen die huidige probleem en vorige probleme wat hulle opgelos het (Mevarech & Kramarski, 1997:369–370; Sumiga, 2010:68).

Te min deursettingsvermoë

Volgens Gomes en Mendes (2007:2) gee leerders maklik moed op indien hulle nie gou ’n oplossing vir ’n probleem kan kry nie. In plaas daarvan om aan te hou probeer, vra hulle iemand anders of gee moed op.

Begrip van programmeringsbegrippe

’n Verdere struikelblok by probleemoplossing kan toegeskryf word aan die feit dat leerders nie weet hoe bepaalde programmeringsbegrippe of -strukture werk nie, of ’n verkeerde begrip daarvan het. Om hierdie struikelblok te help oorkom, kan leerders voltooide programme analiseer, met inbegrip van programme wat logiese foute en programme wat goeie programmeringsbeginsels bevat. Hulle kan ook gevra word om onvoltooide programme te voltooi (Gomes & Mendes, 2007:2–3).

Terwyl sommige vakgebiede eerstens oppervlakkige leer van bepaalde kennis vereis, wat dan later met diepere leer geanaliseer en toegepas word, vereis die aanleer van programmering, soos bo verduidelik, dat leerders tegelykertyd die reëls van sintaksis van die taal moet memoriseer en dit deur middel van hoër vaardighede, soos algoritme-ontwerp en probleemoplossing, moet kan toepas. Jenkins (2002:54) voer aan dat leerders oor die algemeen met die hoër kognitiewe take sukkel en dit moeilik vind om daarop te fokus, omdat hulle vasgevang raak in die laer kognitiewe take, naamlik sintaksis en semantiek van die taal. Die probleem word versterk wanneer onderwysers tydens die onderrig van programmering hoofsaaklik op die onderrig van die sintaksis van die taal fokus, ten koste van onderrig rakende die ontwerp van algoritmes en probleemoplossing (Gomes & Mendes, 2007:3; Jenkins, 2002:55). Onderwysergesentreerde lesings waartydens sintaksis onderrig word tesame met afgesaagde voorbeelde en opdragte dra nie daartoe by om leerders te motiveer en te inspireer nie (Jenkins, 2002:56). Hierdie soort lesings is vermoeiend vir leerders en verhoog hulle angsvlakke as gevolg van die vele vreemde begrippe

waaraan hulle blootgestel word. Gevolglik ervaar leerders programmering as moeilik om aan te leer (Gao, 2011:1267–1268). Die doel van ’n inleidende programmeringskursus behoort primêr daarop gerig te wees om programmeringsbeginsels en -begrippe by leerders tuis te bring en nie noodwendig om die sintaksis van ’n spesifieke programmeertaal, soos bv. *Delphi*, aan te leer nie (Gomes & Mendes, 2007:2; Jenkins, 2002:55).

Uit die voorafgaande behoort dit duidelik te wees dat programmering ’n baie moeilike aktiwiteit is om aan te leer (Donchev & Todorova 2013:84; Gomes & Mendes, 2007:1) en kompleks is om te onderrig (Bennedsen, 2008:3). Volgens Donchev en Todorova (2013:84) kan die rede hiervoor daaraan toegeskryf word dat onderrigmetodes om dit te onderskryf, nie altyd tydens die onderrig van programmering nagevolg word nie. Behalwe vir die wye omvang van die onderrig van programmering, soos in die vorige paragrawe genoem, vind die onderrig van programmering midde in ’n snel veranderende wêreld van tegnologie plaas (Guglielmino, 2003:23). Die programmeertaal wat nou aangeleer word, is verseker nie die taal waarin die leerder vir die volgende jare noodwendig gaan bly programmeer nie. Onderwysers behoort gevolglik die onderrig van programmering so in te rig dat leerders lewenslange leerders wat by nuwe en veranderende omstandighede kan aanpas, sal bly. Volgens Guglielmino (2003:24) is selfgerigte lewenslange leer noodsaaklik ten einde in ’n veranderende wêreld te oorleef en vooruit te gaan. Onderwysers se onderrig–leerstrategieë bepaal tot ’n groot mate hoe leerders vir aanpassings voorberei word. Daar rus dus ’n groot verantwoordelikheid op onderwysers om die geskikste onderrig–leerstrategie tydens die onderrig van programmering te selekteer (Jenkins, 2002:54).

Vervolgens word strategieë vir die onderrig van programmering van nader beskou.

2.7.1 Strategieë vir die onderrig van programmering

Plutargos het reeds in die eerste eeu gesê dat die menslike brein nie ’n houer is wat gevul moet word nie, maar ’n vuur wat aangesteek moet word (soos verwys na deur Guglielmino, 2003:24). In 2010 is daar beraam dat die hoeveelheid digitale inligting elke vyf jaar tien keer vermeerder (Anon, 2010). Die tradisionele manier van onderrig, naamlik om leerinhoud direk aan leerders oor te dra (Gomes & Mendes,

2007:1; Sams & Bergmann, 2013:16) kan in die lig van die voorafgaande nie meer as voldoende geag word nie. Die voorgestelde onderrig–leerstrategieë gaan van die veronderstelling uit dat leer op verskeie maniere plaasvind, soos onder andere deur middel van leerdergesentreerde aktiwiteite (Sams & Bergmann, 2013:16; Yadin, 2013:15), interaksie met ander leerders en selfs deur interaksie met aanlyngemeenskappe (Siemens, 2004:5). Die fokus is dus nie meer op die onderwyser wat onderrig nie, maar op leerders wat leer en kommunikeer, om onafhanklike, lewenslange leerders te kweek (Robins *et al.*, 2003:156). Onderrig beteken nie dat onderwysers kennis aan leerders moet oordra nie, maar eerder dat verskeie aktiwiteite geskep word, sodat leerders aktief besig is en sodanig gestimuleer word dat hulle self modelle vir prosedurele kennis skep (Yadin, 2013:15).

Gomes en Mendes (2007:1) verskaf die onderstaande redes waarom tradisionele onderrigmetodes nie meer vir leerders se behoeftes voldoende is nie:

- leerders benodig persoonlike onderrig, aangesien hulle ’n behoefte het aan onmiddellike terugvoer op programmeringsprobleme en -begrippe wat hulle nie verstaan nie;
- onderrig–leerstrategieë ondersteun nie alle leerders se leerstyle nie, en leerders moet dikwels by die leerstyl van die onderwyser aanpas; en
- onderwysers konsentreer op die onderrig van die sintaksis van die programmeertaal in plaas daarvan om op probleemoplossing deur middel van ’n programmeertaal te konsentreer.

Vervolgens word moontlike strategieë vir die onderrig van programmering aan die hand van bostaande van nader beskou.

2.7.1.1 Koöperatiewe leer

Tydens koöperatiewe leer word minder tyd aan tradisionele lesings bestee en meer tyd aan aktiewe leer. Leerders werk in groepe saam om probleme op te los, deel hulle idees, bespreek probleme wat hulle met die werk ervaar en leer by mekaar (Johnson *et al*., 2008:2:11). Leerders het gewoonlik vrymoedigheid om in hulle klein groepies aan besprekings deel te neem en voel bemagtig omdat hulle nie alleen met probleme worstel nie. Metakognisie verhoog ook in koöperatiewe omgewings,

aangesien die bespreking van probleme tot die begrip daarvan bydra en tot nuwe idees lei (Mevarech & Kramarski, 1997:369; Sandi-Urena *et al*., 2010:15).

Verskeie navorsingstudies het bewys dat koöperatiewe leer ’n suksesvolle onderrig–leerstrategie vir programmering is. ’n Studie van Bagley en Chou (2007:215) het aangetoon dat die waarde van koöperatiewe leer toeneem met die kompleksiteit van probleme. Hulle beveel aan dat koöperatiewe leer veral by die formulering van probleme en die ontwerp van oplossings toegepas word. Beck en Chizhik (2008:207) het in hulle navorsing bewys dat koöperatiewe leer ook ten opsigte van verskillende etniese groepe en veral onder meisies ’n suksesvolle onderrig–leerstrategie vir programmering is. Hulle navorsing het bevind dat studente wat deur middel van ’n koöperatiewe leerstrategie in programmering onderrig is, ’n aansienlike verbetering in prestasie getoon het. ’n Verdere bevinding van Beck en Chizhik (2008) se navorsing was dat die verbeterde prestasie van studente aan die onderrig–leerstrategie self toegeskryf kan word en nie aan die persoon wat die onderrig gegee het nie.

Kritiek wat soms teen koöperatiewe leer geopper word, is dat minder tyd aan lesings bestee word. Lahtinen *et al*. (2005:17) stel egter juis voor dat leerders meer tyd aan aktiewe programmering as aan lesings moet bestee ten einde die probleme wat beginnerprogrammeerders ervaar, die hoof te bied.

Die onderrig–leerstrategieë paarprogrammering en probleemgebaseerde leer, wat op koöperatiewe leer gebaseer is en waarvan die sukses tydens die onderrig van programmering reeds deur navorsing bewys is, word vervolgens bespreek.

2.7.1.2 Paarprogrammering

Paarprogrammering is wanneer twee programmeerders aan een program werk met behulp van slegs een rekenaar. Die programmeerder wat die kode intik, word die *drywer* genoem. Die ander programmeerder word die *navigator* genoem en sy/haar rol is om te kyk wat getik word, foute raak te sien en seker te maak dat die probleem reg opgelos word (Williams *et al*., 2002:197–198). Die navigator is deurgaans op die hoogte van die program en kan onmiddellik as drywer oorneem wanneer die rolle geruil word. Al word verskillende rolle deur die persone betrokke by die paarprogrammeringsproses beoefen, het Bryant *et al.* (2008:527) se navorsing bewys dat albei persone op dieselfde vlak van abstraksie werk en nie een van die

partye benadeel word oor die rol wat hy/sy vervul nie. As gevolg van die komplekse aard van programmering dra die interaksie tussen die programmeerders, die verbalisering van abstrakte begrippe en die onderskeie rolle wat elke programmeerder vul, by tot die sukses van hulle oplossing.

Die gesprekke wat ten opsigte van abstrakte begrippe en die oplossing van probleme ontstaan, kan as een van die belangrikste faktore by die sukses van paarprogrammering gesien word (Bryant *et al.,* 2008:519; Williams *et al*., 2002:198), aangesien navorsing bewys het dat, wanneer ’n sekere begrip aan iemand verduidelik word, dit tot diepere begrip daarvan lei (Chi *et al.,* 1994:439; Williams *et al*., 2002:199). Die gesprekke tussen die paar help dus om die kompleksiteit van die taak te verlig (Bryant *et al.,* 2008:527). Uit die voorafgaande blyk dit dat ’n diepere insig in programmeringsbegrippe en die toepassing daarvan, en verligting van die kognitiewe lading van abstraksie tydens die oplossing van programmeringsprobleme, van paarprogrammering ’n geskikte onderrig–leerstrategie tydens die onderrig van programmering maak (Williams *et al*., 2002:210).

Volgens Williams *et al*. (2002:208) speel die onderwyser ’n belangrike rol in die sukses van paarprogrammering. Die onderwyser moet eerstens die doel van paarprogrammering gereeld aan leerders verduidelik. Verder is die teenwoordigheid van die onderwyser belangrik tydens toesig oor die aktiwiteite en veral om te verseker dat die rolle van drywer en navigator gereeld omgeruil word.

Mentz *et al*. (2008) het paarprogrammering met vyf beginsels wat deur Johnson *et al*. (2008:1:31) vir suksesvolle koöperatiewe leer aanbeveel word, gekombineer en het in hulle navorsing bevind dat, indien hierdie beginsels tydens paarprogrammering toegepas word, goeie resultate tydens die onderrig van programmering verkry word. Die beginsels van koöperatiewe leer en die toepassing daarvan in paarprogrammering word soos volg gestel (Mentz *et al.,* 2008:249–250):

Positiewe interafhanklikheid

Elke lid van die paar moet besef dat hy of sy slegs kan slaag as die ander persoon ook slaag. Doelwitte moet gestel word en beloning vir goeie werk moet sodanig aangebied word dat beide verantwoordelik sal wil wees vir suksesvolle bereiking

daarvan. Die onderwyser moet toesig hou dat die rolle van drywer en navigator geruil word.

Individuele verantwoordelikheid

Maatreëls moet in plek wees om te verseker dat elke lid van die paar ewe veel deel het aan die oplossing van die program of uitvoering van die taak. Enige een van die paar kan byvoorbeeld deur die onderwyser gevra word om die oplossing, of ’n deel daarvan, te verduidelik. Individuele assessering word voorgestel, waartydens soortgelyke probleme individueel opgelos moet word.

Bevordering van persoonlike interaksie

Die paar werk saam aan ’n oplossing en moet verstaan hoe die rolle van drywer en navigator werk. Hierdie rolle moet gereeld geruil word sodat albei lede van die paar ervaring in al twee rolle opdoen. Oplossings sowel as probleme moet met mekaar bespreek word. Dit sal egter slegs sinvol gedoen kan word indien die paar begryp wat elkeen se verantwoordelikheid is en hoe hulle moet saamwerk om ’n gemeenskaplike doelwit te bereik. Die paar is dus vir mekaar se leer verantwoordelik en hulle gemeenskaplike doelwit is dat elkeen die werk na die beste van sy/haar vermoë bemeester.

Sosiale vaardighede

Die paar moet duidelik met mekaar kommunikeer, leer om mekaar te kritiseer sonder om te affronteer, en mekaar aanmoedig en komplimenteer. Die rolle van drywer en navigator moet met gereelde tussenposes geruil word om die onderskeie vaardighede na wense te ontwikkel. Die verbetering van sosiale vaardighede behoort ook die genot van samewerking en motivering te verhoog.

Groepnadenke

Die paar moet gereeld oor hulle vordering en hoe hulle kan verbeter, reflekteer. Breed (2010:246) beveel aan dat leerders ook in metakognitiewe strategieë onderrig word om die leerwins verder te verhoog.

Ten spyte van die voordele wat paarprogrammering as onderrig–leerstrategie inhou, het navorsing aangetoon dat dit nie algemeen in skole in die Noordwesprovinsie toegepas word nie (Mentz & Goosen, 2007:341).

2.7.1.3 Probleemgebaseerde leer

Probleemgebaseerde leer (PBL) is ’n onderrig–leerstrategie, waar leerders aan komplekse probleme uit die werklike wêreld blootgestel word. Hierdeur word hulle aangemoedig om self begrippe en beginsels wat hulle benodig om die probleem op te los, na te vors (Williams, 2004:278). Leerders leer gevolglik om krities te dink en hulle probleemoplossingsvaardighede word ontwikkel. PBL behels dat leerders in klein groepies werk, met mekaar kommunikeer, en saamwerk om bestaande vaardighede te integreer en nuwe vaardighede aan te leer (Duch *et al*., 2001:6). Volgens Norman en Schmidt (1992:559) is die voordele van PBL verhoogde retensie van kennis, bevordering van oordrag van begrippe na nuwe probleme, verhoging in belangstelling in die vakinhoud en bevordering van selfgerigte leervaardighede.

Nuutila *et al.* (2005:124) het Schmidt (1983) se sewe-stap-PBL-metode vir die onderrig van rekenaarprogrammering aangepas en bewys dat dit ’n suksesvolle onderrig–leerstrategie is vir die onderrig van beginnerprogrammeerders. Nuutila *et al.* (2005:124) noem dat probleme wat leerders met die aanleer van begrippe ervaar, vinnig deur onderwysers met PBL geïdentifiseer kan word as gevolg van die noue interaksie tussen leerders en die onderwyser. Die onderwyser se rol is dié van toesighouer en tutor, wat die besprekings van groepe deurgaans moet monitor en die belangrikheid van selfstudie aan leerders moet beklemtoon (Nuutila *et al*., 2005:128). Indien ’n onderwyser agterkom dat ’n groep nie goed funksioneer nie of die spoor byster raak, moet hy/sy die groep bystaan deur byvoorbeeld toepaslike vrae aan die groep te vra. Die onderwyser moet egter daarteen waak om té aktief by die groep betrokke te raak, sodat hulp nie die vorm van ’n mini-lesing aanneem en die groep die verantwoordelikheid om self te leer na die onderwyser verplaas nie (Nuutila *et al*., 2005:126).

Elke leerder moet individueel ook al die begrippe bestudeer om die doelwitte te bereik, aangesien programmering ’n vaardigheid is wat aangeleer moet word en leerders nie net sommige begrippe kan bestudeer nie. Om leerders te help met

bemeestering van begrippe, moet die onderwyser deurlopend individuele oefeninge verskaf, wat spesifiek op die aanleer van begrippe gerig is. Indien leerders nie hierdie stap ernstig opneem nie, ontstaan die gevaar dat PBL nie suksesvol gaan wees nie (Nuutila *et al*., 2005:128).

Behalwe vir bogenoemde onderrig–leerstrategieë wat tydens die onderrig van programmering gevolg kan word, kan die onderrig van programmering self vanuit verskillende benaderings geskied.

2.7.2 Objekte-eerste- versus objekte-laaste-benadering

Die onderrig van ’n SOOP, soos *Delphi* of *Java*, kan vanuit een van twee benaderings geskied, naamlik ’n *objekte-eerste-benadering* of ’n *objekte-laaste-benadering* (Bergin, 2009:1; Bruce, 2004:29). Die objekte-eerste-benadering behels dat leerders van die begin af aan objekgeoriënteerde begrippe en beginsels blootgestel word en dat hulle dadelik vanuit ’n objekgeoriënteerde benadering (sien 2.5.2) begin programmeer (Bergin, 2009:3). Met ’n objekte-laaste-benadering word onderrig eers vanuit ’n prosedurele benadering gegee (sien 2.5.2) en word daar later oorgeskakel na ’n objekgeoriënteerde benadering. Die implikasies van beide hierdie benaderings op die onderrig van programmering word vervolgens bespreek.

Sajaniemi en Kuittinen (2008:76) voer aan dat die aanleer van programmering die belangrikste kwessie moet wees en nie die navolg van ’n bepaalde benadering nie. Volgens hulle bestaan daar feitlik geen teorieë oor die ontwikkeling van programmeringsvaardighede wat objekgeoriënteerde begrippe betref nie en word nuwe kurrikulums en onderrigmetodes wat objekgeoriënteerde programmering insluit, intuïtief geïmplementeer sonder dat dit behoorlik nagevors is. Hulle gee verskeie voorbeelde van hoe die kompleksiteit van Du Boulay (1989) se eerste drie domeine, naamlik die notasie van die taal, die notasiemasjien en die oriëntering van die taal (sien 2.3), aansienlik toeneem wanneer programmering in ’n SOOP vanuit ’n objekte-eerste-benadering aangeleer moet word, teenoor ’n prosedurele benadering. Volgens hulle is daar ’n groot aantal moeilike begrippe in die notasie van ’n SOOP wat nie vir die oplossing van ’n probleem benodig word nie, maar slegs daar is ter wille van die struktuur van die taal (Sajaniemi & Kuittinen, 2008:78). Objekgeoriënteerde programmering behels ook ’n komplekse notasie-masjien wat leerders moet begryp

as gevolg van die insluiting van objekgeoriënteerde begrippe. Selfs met aanvangsonderrig in 'n SOOP vanuit 'n objekte-eerste-benadering moet leerders 'n baie ingewikkelde notasiemasjien verstaan (Sajaniemi & Kuittinen, 2008:80). Wat oriëntasie betref, het Sajaniemi en Kuittinen (2008:81) bevind dat leerders uit 'n prosedurele benadering programme skryf wat betekenisvolle bewerkings doen, maar dat leerders by 'n objekte-eerste-benadering op die skryf van konseptuele modelle vir die data moet konsentreer. Gevolglik bestee leerders meer tyd daaraan om objekte te verstaan en minder tyd aan probleemoplossing (Sajaniemi & Kuittinen, 2008:81). Sajaniemi en Kuittinen (2008:84) bepleit dus 'n prosedurele benadering, aangesien hulle aanvoer dat die kognitiewe lading wat tydens 'n objekgeoriënteerde benadering op leerders geplaas word, net te groot is.

In teenstelling met 'n objekte-laaste-benadering is die voordeel van 'n objekte-eerste-benadering dat die paradigmaskuif vanaf prosedureel na objekgeoriënteerd nie gemaak hoef te word nie (Bergin, 2009:4; Bruce 2004:31). Dit word egter om 'n paar redes as 'n uitdaging beskou om 'n objekte-eerste-benadering te volg. Een daarvan is dat leerders en onderwysers gewoonlik so konsentreer om die sintaksis van 'n SOOP te bemeester dat dit leerders se aandag aftrek van die onderliggende objekgeoriënteerde begrippe en beginsels (Börstler *et al.*, 2008:81). 'n Tweede rede is dat daar min beskikbare hulpmiddels en pedagogiese strategieë is om 'n SOOP volgens hierdie benadering te onderrig (Bruce, 2004:32; Sajaniemi & Kuittinen, 2008:76) en dat hierdie eksterne faktore, sowel as om leerders as gevolg van die moeilikheidsgraad gemotiveerd te hou, juis onderrig bemoeilik (Kölling, 2008:100).

Die kwessie of 'n objekte-eerste of objekte-laaste-benadering in programmering gebruik moet word, is 'n kontroversiële vraag waaroor baie gedebatteer word (Astrachan *et al*., (2005:451). Uit bostaande is dit duidelik dat teenstrydige opinies oor die keuse van 'n benadering vir die onderrig van 'n SOOP bestaan. Dit is egter buite die veld van die studie om 'n spesifieke benadering voor te stel, maar wel om ondersoek in te stel na die onderrig van objekgeoriënteerde begrippe.

2.7.3 Die onderrig van objekgeoriënteerde begrippe

Basiese objekgeoriënteerde begrippe soos klasse, objekte en verwantskappe is op 'n hoë vlak van abstraksie en moeilik om te onderrig (Bruce, 2004:32; Kölling,

2008:105). ’n Steil leerkurwe word dus gevolg in die onderrig daarvan (Börstler *et al.*, 2008:81), aangesien die basiese begrippe van programmering verweef is in objekgeoriënteerde programmering en nie in isolasie onderrig kan word nie (Roberts *et al.*, 2006). Dit impliseer dat moeilike begrippe van die begin af deur leerders bemeester en toegepas moet word en dat dit nie altyd moontlik is om eers met eenvoudige begrippe te begin nie (Sajaniemi & Kuittinen, 2008:80).

Hierdie hoë eise wat objekgeoriënteerde programmering stel, het tot gevolg dat alternatiewe onderrig–leerbenaderings benodig word om objekgeoriënteerde begrippe by leerders te vestig (Bruce, 2004:33). Verskeie voorstelle is in die verband gemaak. Een voorstel is om objekgeoriënteerde begrippe konkreet voor te hou deur middel van mikrowêrelde soos *Karel die Robot* en *Alice* (Bruce, 2004:32). ’n Verdere moontlikheid is rolspelaktiwiteite waar leerders objekte met rolspel naboots, amper soos akteurs in ’n toneel (Börstler *et al.*, 2008:85). Daar word ook voorgestel dat goeie voorbeelde van scenario’s en programme aanvanklik aan leerders gegee word, sodat hulle duidelik kan sien hoe programme vanuit ’n objekgeoriënteerde benadering funksioneer (Kölling, 2008:107). Bruce (2004:32) stel ook voor dat klasse geskep word wat leerders interesseer, sodat hulle van die begin af interessante programme kan skryf sonder om te veel op sintaksis te fokus, maar eerder op die styl van objekgeoriënteerde programmering.

Weisfeld (2009:1) voer aan dat objekgeoriënteerde begrippe nie deur die aanleer van ’n spesifieke programmeertaal aangeleer word nie, maar dat ’n bepaalde denkwyse vir objekgeoriënteerde programmering vooraf gevestig moet word. Hy beklemtoon dat leerders aan objekgeoriënteerde begrippe blootgestel moet word voordat hulle met die aanleer van ’n objekgeoriënteerde programmeertaal begin ten einde ’n objekgeoriënteerde denkproses te vestig. Hierdie voorstel kan moontlik ’n antwoord wees op Bruce (2004:32) se soeke na alternatiewe pedagogiese benaderings en Börstler *et al.* (2008:81) se bekommernis dat die fokus op sintaksis die aandag van objekte kan aflei. Die vraag kan ook dus gestel word of dit nie die kompleksiteit van objekgeoriënteerde benadering sal verlaag indien objekgeoriënteerde begrippe en ’n objekgeoriënteerde denkpatroon reeds afgehandel is wanneer met die onderrig van ’n SOOP begin word nie.

Daar word van IT-leerders in Suid-Afrikaanse skole verwag om *Scratch*-programmering in graad 10 te doen en in graad 11 na ’n SOOP oor te skakel. Uit die bespreking van die aard van *Scratch* en die aard van *Delphi* (sien 2.4.2 en 2.5.2) is dit duidelik dat die oorskakeling tussen die programmeertale op sigself ’n groot paradigmaskuif is. Tesame hiermee word *Scratch* nie as ’n objekgeoriënteerde programmeertaal beskou nie en word ’n prosedurele benadering gewoonlik gevolg (sien 2.4.4) teenoor ’n objekgeoriënteerde benadering wat vir *Delphi* voorgeskryf word (DBE, 2011:12). Indien *Scratch*-onderrig nie as ’n geleentheid gebruik word om reeds bepaalde objekgeoriënteerde begrippe by leerders te vestig nie, sal dit beteken dat leerders nog ’n groter paradigmaskuif moet maak (Bergin, 2009:4; Bruce 2004:31) met die oorskakeling na ’n objekgeoriënteerde benadering in *Delphi*. Dit blyk dus dat die uitdaging vir IT-onderwysers is om *Scratch* sodanig te onderrig dat leerders reeds blootstelling aan objekgeoriënteerde begrippe ontvang het voordat hulle met *Delphi*-programmering begin.

Vervolgens word die onderrig van *Scratch* ten opsigte van moontlike onderrig–leerstrategieë en voor- en nadele met betrekking tot die aanleer daarvan as programmeertaal in oënskou geneem.

2.8 DIE ONDERRIG VAN *SCRATCH*

Scratch is ontwerp met tieners, wat volgens Malan en Leitner (2007:227) nie graag uit handboeke werk of van lesings hou nie, in gedagte. Omdat tieners gewoonlik vol afwagting met *Scratch* kennis maak en so gou moontlik wil begin programmeer, is die ideaal om *Scratch*-programmering op so ’n wyse te onderrig dat beginnerprogrammeerders vinnig op hulle voete kom, en so gou moontlik kan begin programmeer (Ford, 2009:xv).

Sodra die GOO aan leerders bekend gestel is en hulle weet hoe om blokke in te sleep, kan hulle begin eksperimenteer en *Sprites* aksies laat uitvoer (Malan & Leitner, 2007:224). Hiermee word nie bedoel dat eksperimentering die primêre strategie vir die onderrig van *Scratch* is nie; dit beklemtoon eerder dat leerders graag met *Scratch* eksperimenteer (Meerbaum-Salant *et al.,* 2013:240; Resnick *et al.*, 2009:60; Wolz *et al.*, 2008:298). Clear (1997:25) het reeds in 1997 ’n interaktiewe, ondersoekende aanslag vir die onderrig van programmering voorgestel, maar op

daardie stadium was daar nog nie visuele programmeertale soos *Scratch* beskikbaar vir die onderrig van programmering nie. Daar moet ook in gedagte gehou word dat *Scratch* ontwikkel is met die doel om leerders se belangstelling in programmering op ’n informele manier te prikkel, met minder fokus op direkte onderrig (Maloney *et al*., 2010:2).

2.8.1 ’n Onderrig–leerstrategie vir *Scratch*

Uit bostaande bespreking is dit duidelik dat ’n ander strategie vir die onderrig van *Scratch*-programmering gevolg moet word as die tradisionele, lesinggebaseerde onderrig–leerstrategie. Wolz *et al*. (2009:2) meen dat *Scratch*-onderrig ’n kombinasie moet wees van eksplorasie en spel. Minder laevlak- kognitiewe take soos die aanleer van sintaksis hoef onderrig te word (Jehng *et al.*, 1999:288) en die fokus kan op die aanleer van die kuns van programmering geplaas word (Wolz *et al*., 2009:3).

Wang en Zhou (2011:489-491) het ’n onderrigbenadering voorgestel om programmeringsbegrippe in *Scratch* aan te leer vir latere toepassing in ’n tradisionele programmeertaal. Hulle meld egter nie wat die tradisionele programmeertaal is nie. Hulle onderrigbenadering met *Scratch* fokus daarop dat ’n bekende situasie as leeromgewing gebruik moet word. Hulle stel voor dat bekende, opwindende situasies en pret-programmeringsomgewings gebruik moet word om programmering aan hoërskoolleerders te onderrig, sodat leerders se angstigheid verminder en hulle belangstelling om te leer geprikkel word. In die benadering wat hulle voorstel, word ’n probleem in ’n bekende situasie as vertrekpunt gebruik. Die bepaalde situasie moet ontleed word en ’n algoritme moet vir die oplossing van die probleem ontwerp word voordat ’n *Scratch*-program geskryf word om die probleem op te los.

Alhoewel *Scratch* ’n pret-omgewing is wat selfgerigte leer en selfontdekking aanmoedig (Meerbaum-Salant *et al*., 2011:168), is *Scratch* ook ’n gesofistikeerde stelsel (sien 2.4.4) waarmee programmeringsbeginsels en –begrippe reeds aangeleer kan word.

Malan en Leitner (2007:223) het ’n kwalitatiewe studie gedoen om te bepaal of programmeringsbeginsels suksesvol deur middel van *Scratch* aan studente bekend gestel kan word voordat hulle na programmering in die SOOP *Java* oorgaan. Studente is een week in *Scratch* onderrig en sewe weke in *Java*. In die eerste

Scratch-les is die mees fundamentele programmeringsbegrippe, naamlik stellings, Boole-uitdrukkings, veranderlikes, besluitneming en herhaling onderrig. Die studie verwys nie pertinent na die onderrig–leerstrategie wat gevolg is nie, maar die afleiding kan gemaak word dat nuwe begrippe in kort lesings verduidelik is, gevolg deur probleemscenario's wat studente individueel moes oplos. Programmeringsbegrippe is aanvanklik deur middel van probleme uit die werklike lewe gedemonstreer, waarvan die oplossings eerstens met pseudokode voorgestel is om die begrippe te beklemtoon en te verduidelik, voordat *Scratch*-oplossings daarvoor geskep is. Daarna is rygskakels en gebeurtenisse aan studente verduidelik. Studente moes daarna hulle eie probleemscenario's uitdink en *Scratch*-oplossings daarvoor ontwerp. Die enigste voorwaardes wat vir die probleemscenario's gestel is, was dat bepaalde programmeringsbegrippe, soos deur die dosent gegee, daarin moes voorkom. Daarmee is gepoog om studente se nuuskierigheid en kreatiwiteit na vore te bring, in plaas daarvan om hulle in te perk met spesifieke opdragte (Malan & Leitner, 2007:225). In die tweede les is studente se voltooide projekte van nader beskou om sodoende blootstelling te gee aan die lees en begrip van programmeringskode. Die les is afgesluit met 'n voorskou van soortgelyke programme in *Java*-kode. Die daaropvolgende opdrag het behels dat studente moes voortbou op ander studente se projekte. Voltooide opdragte is aan die klas voorgelê en studente moes daaroor reflekteer. Ná afloop van die studie het 76% van die studente gesê dat *Scratch* hulle gehelp het om programmeringsbegrippe wat hulle in *Java* moes gebruik, beter te verstaan. Hulle het beweer dat dit aan hulle 'n aanduiding gegee het hoe programmeerders dink en hoe om programmeringsprobleme te benader wanneer hulle met *Java* begin het. Hulle het ook beweer dat hulle intuïtief verstaan het hoe herhaling en veranderlikes in *Java* werk (Malan & Leitner, 2007:227).

Meerbaum-Salant *et al*. (2013:239) het met 'n gemengde-metode studie onder hoërskoolleerders van ongeveer 15 jaar oud ondersoek ingestel of programmeringsbeginsels deur middel van *Scratch* aangeleer kan word. Twee klasse is deur verskillende onderwysers gedurende gewone klastyd onderrig. Die een onderwyser het 15 jaar ondervinding in die onderrig van programmering gehad en die ander onderwyser was 'n Wiskunde-onderwyser sonder enige ondervinding in die onderrig van programmering. Dit was egter beide onderwysers se eerste

kennismaking met *Scratch*. Die navorsers het ’n handboek ontwikkel wat leerinhoud moes leer en projekte in detail beskryf het. Die handboek was vir gebruik deur die onderwysers en is nie aan leerders beskikbaar gestel nie (Meerbaum-Salant *et al*., 2011:169). Die navorsers het ’n konstruktivistiese onderrigfilosofie gevolg, waar leerders nie uit handboeke moes werk nie, maar self oplossings moes konstrueer. Alhoewel die ervaring van die onderwysers wat leerders tydens die studie in programmering onderrig het, baie verskil het, is bevind dat daar nie ’n beduidende verskil tussen die prestasies van leerders van die onderskeie klasse was nie (Meerbaum-Salant *et al*., 2013:243). Die studie het verder bevind dat sekere programmeringsbegrippe wel suksesvol deur middel van *Scratch* aangeleer kan word. Leerders het ’n goeie begrip van onvoorwaardelike- en voorwaardelike lusstrukture sowel as die oorstuur van boodskappe getoon, maar nie inisialisering, veranderlikes en gelyklopendheid nie (Meerbaum-Salant *et al*., 2013:245). Laasgenoemde is deur Meerbaum-Salant *et al*. (2013:263) aan die onderrigproses toegeskryf en daar word beweer dat hierdie programmeringsbeginsels ook deur middel van *Scratch* aangeleer kan word indien behoorlike onderrigmetodes toegepas word.

Die volgorde waarop Meerbaum-Salant *et al*. (2013:240) besluit het om die programmeringsbegrippe te onderrig, was gebaseer op die bevindings van Maloney *et al.* (2008:370). Laasgenoemde het leerders tussen die ouderdomme 8 en 18 jaar in ’n naskoolsentrum aan *Scratch* blootgestel, om te bepaal watter programmeringsbegrippe hulle self kon ontdek. Die leerders het geen formele programmeringsonderrig ontvang nie en het teen hulle eie tempo aan projekte van hulle keuse gewerk. Daar was wel mentors wat hulle om hulp kon vra wanneer nodig. Die bevindings van die studie was dat alle leerders *Script*s ontwikkel het wat sekwensiële uitvoering gebruik. Van die leerders het 88% gelyklopende *Script*s gebruik, 53,6% het toevoer- of afvoer gebruik, 51,8% het herhaling gebruik, 26,1% het voorwaardelike instruksies gebruik, 24,7% het kommunikasie en sinchronisasie gebruik, 10,8% het Boole-logika gebruik, 9,6% het veranderlikes gebruik en 4,7% het willekeurige getalle gebruik (Maloney *et al.,* 2008:3). Die gevolgtrekking is gemaak dat die programmeringsbegrippe met ’n hoë gebruikspersentasie maklik deur leerders self ontdek kan word en programmeringsbegrippe met ’n lae persentasie gewoonlik nie deur leerders self ontdek word nie. Begrippe wat ’n lae persentasie

behaal het, was veranderlikes, Boole-logika en willekeurige veranderlikes. Maloney *et al.* (2008:3) beveel aan dat meer leiding aan leerders betreffende dié begrippe verskaf moet word.

Meerbaum-Salant *et al*. (2013:240) het programmeringsbegrippe in die volgende volgorde onderrig: gelyklopendheid deur die gebruik van meervoudige *Sprites*, gelyklopendheid deur die gebruik van meervoudige *Script*s vir een *Sprite*, kommunikasie en sinchronisasie deur die stuur en ontvang van boodskappe, herhaling, veranderlikes, voorwaardelike uitvoering van gebeurtenisse, numeriese bewerkings, en laastens tellers en skikkings. In plaas van om spesifieke *Scratch*-eienskappe te onderrig, is die onderrig so beplan dat bepaalde programmeringsbegrippe onderrig is. Elke hoofstuk in die handboek, wat die volgorde van onderrig bepaal het, was gebaseer op ’n sekere projek wat bepaalde programmeringsbegrippe bevat het. Programmeringsbegrippe is aan leerders bekend gestel soos wat dit nodig geag is om die projekte te voltooi (Meerbaum-Salant *et al*., 2013:240). Ruimte is gelaat vir leerders wat projekte anders benader het, sodat die kreatiwiteit van leerders nie ingeperk is nie. Aspekte soos kostuums van *Sprites* en visuele effekte is nie in die projekte beklemtoon nie, sodat leerders se aandag nie van bepaalde programmeringsbegrippe wat aangeleer moes word, afgetrek is nie (Meerbaum-Salant *et al*., 2013:240).

Die kwantitatiewe been van bogenoemde studie het behels dat leerders ’n vooraftoets, tussentydse toets en natoets moes aflê (Meerbaum-Salant *et al*., 2013:239). Die toetse is ontwerp om te bepaal wat leerders se begrip van bepaalde programmeringsbegrippe was. Die kwalitatiewe been van die studie het onderhoude met onderwysers en leerders, observasie en ontleding van leerders se *Scratch*-projekte behels. Meerbaum-Salant *et al*. (2013:243) verduidelik die bevindings dat inisialisering, veranderlikes en gelyklopendheid nie so goed begryp is nie deur aan te voer dat dit abstrakte begrippe is wat moeilik is om te begryp. In teenstelling hiermee het Maloney *et al.* (2008:3) se studie bevind dat gelyklopendheid die meeste deur leerders in projekte gebruik is. Maloney *et al.* het dit geïnterpreteer as ’n begrip wat leerders intuïtief verstaan. Wat inisialisering betref, beweer Meerbaum-Salant *et al*. (2013:244) dat leerders dit met die aanvangsuitvoering van ’n *Script* verwar het, naamlik om op die groen vlaggie te klik. Hulle verklaar hierdie verwarring van die woord ‘inisialisering’ as ’n spreektaalkwessie, aangesien leerders moontlik wel

verstaan het hoe inisialisering werk, maar dat die term nie noodwendig in die onderrig gebruik is nie (Meerbaum-Salant *et al*., 2013:244).

Meerbaum-Salant *et al*. (2013:244) skryf die goeie begrip van herhaling daaraan toe dat eenvoudige, onvoorwaardelike herhalingstrukture vroeg reeds aan leerders bekend gestel is en dat dit leerders gehelp het om voorwaardelike herhalingstrukture ook te verstaan. Hulle beweer dat die begrip van veranderlikes in *Scratch* ooreenstem met dié van ander programmeertale. Nie een van die onderwysers het veranderlikes egter eksplisiet onderrig nie en leerders het nie ’n goeie begrip van veranderlikes gehad nie. Meerbaum-Salant *et al*. (2013:244) noem dat dit onwaarskynlik is dat veranderlikes self deur leerders op ’n probeer-en-tref-manier van programmering aangeleer sal word, en kom tot die gevolgtrekking dat veranderlikes eksplisiet onderrig moet word. Maloney *et al.* (2008:3) het dieselfde gevolgtrekking in hulle studie oor veranderlikes gemaak. Malan en Leitner (2007:227) se studie spesifiseer nie wat hulle studente se eerste ervaring met die begrip van veranderlikes in *Scratch* was nie, maar noem wel dat studente dit later in *Java* intuïtief begryp het. Daar moet egter gemeld word dat die deelnemers aan Malan en Leitner (2007:223) se studie studente en nie hoërskoolleerders was nie, wat moontlik kan verklaar waarom die begrip van veranderlikes makliker begryp is.

Yadin (2013:16) het ’n studie onderneem in ’n poging om die slaagsyfer onder eerstejaarstudente in programmering te verhoog. Programmering is vanuit ’n prosedurele benadering onderrig om studente op die daaropvolgende kursus voor te berei, wat ’n objekte-eerste-benadering sou volg. ’n Visuele, objekgeoriënteerde omgewing, *Karl die Robot*, is gelyklopend met die onderrig van die programmeertaal *Python* gebruik, om abstrakte begrippe van programmering konkreet aan studente voor te hou. Die oogmerk was om op die ontwikkeling van algoritmes en probleemoplossingsvaardighede te konsentreer en nie op die onderrig van sintaksis nie. Aanvanklik het die studie nie goeie resultate opgelewer nie en die slaagsyfers het nie verbeter nie. Een van die redes wat genoem word, was dat die onderrig van *Python* op ’n prosedurele benadering gebaseer was, maar dat *Karel die Robot* ’n objekgeoriënteerde benadering gebruik het, en dit het verwarring onder studente veroorsaak (Yadin, 2013:17). Nadat die *Karel die Robot* met ’n visuele omgewing wat ook op ’n prosedurele benadering gebaseer was vervang is, het die studente se punte aansienlik verbeter.

Dit blyk uit die voorafgaande bespreking dat leerders nie op hulle eie alle programmeringsbegrippe kan aanleer nie. Meerbaum-Salant *et al*. (2011:168) beweer dat programmeringsbegrippe slegs behoorlik aangeleer kan word indien dit eksplisiet onderrig en verduidelik word. Hulle beklemtoon dat *Scratch* as ’n hulpmiddel beskou moet word om goeie programmeringsbegrippe aan te leer en dat goeie onderrig–leerstrategieë en leermateriaal steeds benodig word om leer te bewerkstellig. Onderwysers moet daarop let dat begrippe tydens die onderrigsituasie by die naam genoem moet word om bewustheid daarvan by leerders te kweek (Meerbaum-Salant *et al*. 2013:244). Dit kan ook afgelei word uit Yadin (2013:10) se navorsing dat ’n objekgeoriënteerde benadering sover moontlik reeds in *Scratch* gevolg moet word om die verwarring onder leerders uit te skakel wanneer na *Delphi* oorgeskakel word.

Vervolgens word die voor- en nadele van *Scratch* as programmeertaal onder die loep geneem. Die invloed van *Scratch* op die vestiging van programmeringsbegrippe en programmeringsgewoontes gaan ook ondersoek word.

2.8.2 Voor- en nadele van die gebruik van *Scratch* as programmeertaal binne ’n formele onderrigomgewing

Alhoewel daar met die ontwerp van *Scratch* gepoog is om ’n pret-omgewing vir programmering beskikbaar te stel en daar talle voordele aan *Scratch*-onderrig is, het navorsing ook verskeie swak gewoontes wat leerders met *Scratch*-programmering kan ontwikkel, blootgelê.

2.8.2.1 Voordele van die gebruik van Scratch as programmeertaal

Scratch is ’n pret-omgewing om programmering aan leerders te onderrig (Meerbaum-Salant *et al.*, 2013:239). *Scratch* maak leerders opgewonde (Malan & Leitner, 2007:226) en hulle word feitlik met die eerste kennismaking onmiddellik by aktiwiteite wat hulle belangstelling prikkel en by hulle ervaringswêreld aansluit, betrek, soos die skep van animasie, stories en speletjies (Fincher *et al.*, 2010:192–193). Hierdie positiewe ervarings dra by tot ’n beter selfbeeld ten opsigte van programmeringsvermoë, laer angsvlakke (Kapsimali & Sampson, 2011:186; Malan & Leitner, 2007:223; Meerbaum-Salant *et al.*, 2013:239) en verhoog leerders se belangstelling in programmering (Wang & Zhou, 2011:488).

Die visuele terugvoer wat *Scratch* verskaf, is ’n bate tydens die onderrig van programmering (Dee, 2013). Jehng *et al.* (1999:288) het reeds in die vorige eeu in ’n studie bevind dat ’n visuele voorstelling leerders help om ’n akkurate geheuemodel van abstrakte programmeringsbegrippe te vorm. Begrippe wat gewoonlik in programmeertale verskans is en vir beginnerprogrammeerders moeilik is om te verstaan, soos veranderlikes en die uitvoering van programme, word visueel en konkreet uitgebeeld en is dus makliker verstaanbaar in *Scratch* (Wolz *et al.*, 2008:298). Jehng *et al.* (1999:288) beweer dat teksgebaseerde programmeertale dit veral vir beginnerprogrammeerders moeilik maak om ’n begrip te vorm van wat hulle lees en ’n hoë kognitiewe lading verg. Met visuele voorstellings kan meer geheue gereserveer word vir hoër kognitiewe bewerkings en kan ’n diepere verwerking van kennis plaasvind sodat leerervarings uitgebrei kan word en probleemoplossing kan verbeter (Jehng *et al.*, 1999:288).

Aangesien daar geen kwessies oor sintaksis in *Scratch* bestaan nie, kan meer aandag gegee word aan die onderrig van probleemoplossing, algoritme-ontwerp, en die ontwikkeling van programmeringsbegrippe, sonder ’n oorbeklemtoning van die sintaksis van die taal, wat algemeen ’n struikelblok by die ontwikkeling van oplossings is (Deek, 1999:1; Jenkins, 2002:55; Wolz *et al.*, 2008:298).

Alhoewel dit uit die voorafgaande blyk asof *Scratch* voordelig is tydens die onderrig van programmering, is daar ook gevaartekens waarop gelet moet word.

2.8.2.2 Swak programmeringsgewoontes wat met Scratch-programmering gevorm word

Meerbaum-Salant *et al*. (2011:172) het in ’n studie bevind dat leerders geneig is om swak programmeringsgewoontes met *Scratch* aan te leer, juis as gevolg van die eksperimentele aard van *Scratch*. Die outeurs spreek die bekommernis uit dat die aanleer van *Scratch* die aanleer van programmering kan benadeel aangesien gewoontes wat in *Scratch* aangeleer word, verskil van aanvaarde programmeringspraktyke (sien 2.3) (Meerbaum-Salant *et al*., 2011:168). Die kriteria gestel vir ’n gewoonte was eerstens dat dit in baie leerders se programme moes voorkom en herhaaldelik in ’n spesifieke leerder se program moes voorkom, en tweedens dat die leerders dit onbewustelik moes doen, sonder dat hulle dit wou

regverdig of sonder dat hulle alternatiewe daarvoor oorweeg het (Meerbaum-Salant *et al*., 2011:169). Daar is ook bevind dat leerders twee swak programmeringsgewoontes met die aanleer van *Scratch* gevorm het, naamlik dat hulle ekstreme onder-na-bo-programmering en ekstreme bo-na-onder-programmering toegepas het. Bo-na-onder-programmering beteken dat ’n program in kleiner, logiese eenhede verdeel word, wat maklik ontfout en in stand gehou kan word (sien 2.4.5.1). Indien ’n onder-na-bo-benadering vir die oplos van probleme korrek toegepas word, beteken dit dat komponente saamgevoeg word totdat ’n volledige oplossing gevorm is (sien 2.4.5.1).

Ekstreme onder-na-bo-programmering

Détienne (2010:49) voer aan dat beginnerprogrammeerders verkies om eerder ’n onder-na-bo-benadering te gebruik en nuwe oplossings op ’n terugwaartse manier te konstrueer, teenoor ervare programmeerders wat gewoonlik ’n bo-na-onder-benadering gebruik. In *Scratch* voer leerders egter onder-na-bo-ontwerp op ’n ekstreme vlak uit. Volgens Meerbaum-Salant *et al*. (2011:169) het leerders oplossings vir probleme nie met algoritme-ontwerp benader nie, maar verskeie blokke wat lyk of dit kan bydra tot die oplossing van die probleem ingesleep en inmekaar gepas, totdat ’n oplossing vir die probleem gevorm is.

Hulle gevolgtrekking is dat leerders nie met inagneming van algoritmiese ontwerp dink, of beginsels toepas wat in programontwerp benodig word nie en beweer dat hierdie gewoonte deur die *Scratch*-omgewing aangemoedig word. Leerders hoef, volgens hulle, nie instruksies te memoriseer of doelbewus te beplan watter instruksies hulle programme gaan benodig nie aangesien die instruksies heeltyd op die palet in die *Scratch*-GOO beskikbaar en sigbaar is. Blokke wat nie in *Scripts* benodig word nie, kan ook in die *Script*-area rondlê vir ingeval dit later benodig gaan word, sonder dat dit die uitvoering van programme beïnvloed. Alhoewel laasgenoemde bydra tot die interaktiewe konstruksie van *Scripts*, beweer Meerbaum-Salant *et al*. (2011:169) dat dit nie goeie programontwerp aanmoedig nie.

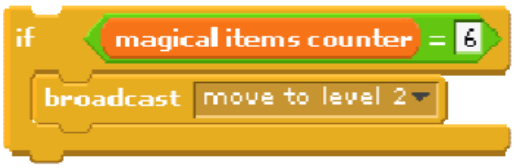
Ekstreme bo-na-onder-programmering

Meerbaum-Salant *et al*. (2011:170) het bevind dat leerders ook ekstreme bo-na-onder-programmeringsgewoontes met *Scratch* kan aanleer en noem dit fyn-vertakte-

programmering. Probleme is nie logies in kleiner probleme opgedeel nie, maar *Scripts* is onnodig in baie klein *Scripts* onderverdeel is, wat nie logies samehangend was nie. Die afleiding word ook deur Meerbaum-Salant *et al*. (2011:170) gemaak dat leerders nie goeie ontwerpbeginsels in die ontwikkeling van programme toegepas het nie.

'n Verdere gevolg van ekstreme bo-na-onder-programmering soos deur Meerbaum-Salant *et al*. (2011:170) aangetoon, was dat leerders besluitnemingstrukture verkeerd gebruik het en die gewoonte ontwikkel het om IF-stellings te vervang met voorwaardelike oneindige herhaling **(**Meerbaum-Salant *et al*., 2011:170) (sien tabel 2.5).

Tabel 2.5: Foutiewe gebruik van besluitnemingstrukture (Meerbaum-Salant *et al*., 2011:170)

IF-stelling	Voorwaardelike oneindige lus
	

'n Verdere uitvloeisel van ekstreme bo-na-onder-programmering, was dat leerders nie begrensde herhalingstrukture gebruik nie, maar eerder kleiner *Script*s om dieselfde programuitvoering te bewerkstellig (Meerbaum-Salant *et al*., 2011:170). Dit was duidelik dat leerders nie 'n grondige begrip van herhalingstrukture gehad het nie, nie geweet het hoe om herhalingstrukture toe te pas nie, en gevolglik slegs verskeie *Script*s gebruik het om die gewenste uitwerking in programuitvoering te kry (sien tabel 2.6). Die gevolg hiervan was ongestruktureerde programme waar goeie programmeringsbeginsels, soos die gebruik van voorwaardelike herhalingstrukture, nie toegepas is nie. Hierdie gevolgtrekking weerspreek hulle vorige bevindings dat leerders herhaling verstaan het (Meerbaum-Salant *et al*. 2013:244). Leerders het dalk moontlik die begrip van herhaling verstaan, maar nie hoe herhaling wat betref goeie programmeringsbeginsels toegepas word nie, en ook nie die toepassing van

voorwaardelike herhalingstrukture nie. Tabel 2.6 toon aan hoe 'n voorwaardelike herhalingstruktuur nageboots word, deur drie afsonderlike *Scripts* te gebruik en swak programmeringsbeginsels dus toegepas word.

Tabel 2.6: Foutiewe opstelling van herhaling (Meerbaum-Salant *et al*., 2011:170)

Voorwaardelike herhalingstruktuur	Drie afsonderlike *Script*s in plaas van die korrekte herhalingstruktuur
	

In Meerbaum-Salant *et al*. (2011:171) se studie het ekstreme bo-na-onder-programmering ook gelyklopende uitvoering van *Script*s gekompliseer. Die *Scratch*-omgewing moedig gelyklopende *Script*s aan omdat verskeie *Sprites* take terselfdertyd moet uitvoer. Te veel gelyklopende *Script*s kan egter veroorsaak dat 'n program nie altyd uitvoer soos wat verwag word nie, en ontfouting van programme bemoeilik (Meerbaum-Salant *et al*., 2011:171).

Volgens Meerbaum-Salant *et al*. (2011:171) ontstaan slegte programmerings-gewoontes juis omdat onderwysers as gevolg van die pret- en leer-deur-te-speel-aard van *Scratch* (sien 2.4.2) nalaat om behoorlike ontwerpbeginsels en goeie programmeringstegnieke soos die gebruik van goeie beheerstrukture en voorwaardelike herhalingstrukture aan leerders te onderrig. Ervare onderwysers wat aan Meerbaum-Salant *et al*. (2011:171) se studie deelgeneem het, het erken dat hulle nagelaat het om bepaalde begrippe en beginsels te onderrig, omdat hulle

gedink het dat leerders dit self met *Scratch* sou ontdek. Hulle het ook aangeneem dat leerders besluitneming en herhaling verstaan het omdat hulle dit in *Scratch*-programme gebruik het en programme geskryf het wat werk. By nadere ondersoek het Meerbaum-Salant *et al*. (2011:172) egter bevind dat leerders nie die strukture korrek verstaan het nie en slegs kodeblokke saamgeflans het totdat hulle werkende programme gekry het. Dit impliseer dat leerders nie vooraf goeie algoritmes ontwerp het en metakognitief oor die gebruik van besluitneming en herhaling gedink het nie. Meerbaum-Salant *et al.* (2011:171) beveel aan dat *Scratch*, soos enige ander programmeertaal, gebruik moet word om goeie programmeringsbeginsels vir leerders aan te leer en waarsku dat hierdie beginsels nie vanself sal ontwikkel nie, maar deur onderwysers gekweek moet word.

Dit word hier gestel dat *Scratch* nie die skuld vir die aanleer van slegte programmeringsbeginsels moet kry nie, aangesien die aanleer van swak programmeringsbeginsels nie in ander studies gerapporteer is nie, maar eerder die teendeel, naamlik dat *Scratch* gehelp het om goeie programmeringsbeginsels te vestig (Malan en Leitner, 2007:227; Wang en Zhou, 2011:491). Dit sou ook onregverdig wees om die blaam op onderwysers te plaas vir die wyse waarop *Scratch* onderrig word, aangesien hulle nie opleiding in die onderrig van *Scratch* ontvang het nie. Kölling (2008:99) het ’n studie onderneem om die uitwerking van die nuwe koppelvlak, *BlueJ*, op die onderrig van studente in *Java* te bepaal. Hy het bevind dat ’n nuwe onderrighulpmiddel nie net aan onderwysers gegee kan word en verwag word dat hulle op hulle eie nuwe onderrigmetodes moet ontdek nie. In Meerbaum-Salant *et al*. (2013:171) se studie was die situasie soortgelyk, aangesien die onderwysers nie vertroud was met *Scratch*-onderrig nie. Die uiteinde was dat swak programmeringsbeginsels deur leerders aangeleer is, alhoewel dit geblyk het dat leerders programmeringsbegrippe verstaan het. Die gevolgtrekking word dus gemaak dat geskikte onderrig–leerstrategieë gekies moet word om die uiteindelike doel van programmeringsonderrig deur middel van *Scratch* te bereik, naamlik om leerders aan die proses van programmering bekend te stel en goeie programmeringsbeginsels aan te leer (Resnick, 2012).

2.9 SAMEVATTING

Ten spyte van ’n snel veranderende wêreld en nuwe programmeertale wat ontwikkel word om met die ontwikkeling van rekenaarapparatuur en -programmatuur tred te hou, het die basiese begrippe van programmering en die aard van programmering oor die jare nie veel verander nie – programmering is steeds ’n komplekse aktiwiteit wat hoëvlakdenke vereis. In hierdie hoofstuk is die aard van programmering en meer spesifiek ook die aard van die programmeertale *Scratch* en *Delphi* ondersoek, met die doel om te bepaal watter programmeringsbeginsels reeds met die onderrig van *Scratch* vasgelê kan word.

Net soos wat programmering ’n komplekse saak is, is die onderrig van programmering nie minder kompleks nie. ’n Geskikte onderrig–leerstrategie moet gevolg word sodat die proses van programmering, en programmeringsbeginsels en -begrippe soos objekgeoriënteerde begrippe wat ’n hoë kognitiewe lading verg, behoorlik onderrig word. Dit is verder ’n omvattende en ingewikkelde pedagogiese kwessie om die proses van programmering op so ’n wyse te ontsluit dat leerders tydens die aanleer daarvan gemotiveerd sal bly.

Scratch en *Delphi* is op die oog af twee uiteenlopende programmeertale. *Scratch* is ’n kleurryke en mediaryke programmeertaal waar programmering in ’n pret-omgewing plaasvind. Alhoewel programmering in *Scratch* gewoonlik volgens ’n prosedurele benadering plaasvind, kan dit ook as ’n objekgebaseerde taal beskryf word. *Delphi* is ’n programmeertaal waar programmeringskode volgens streng sintaktiese reëls ingetik word. Dit is ’n hibriede programmeertaal en programmering kan vanuit ’n prosedurele of objekgeoriënteerde benadering plaasvind. By diepere ondersoek is bevind dat daar egter programmeringsbegrippe is wat in beide *Scratch* en *Delphi* teenwoordig is (sien tabel 2.4). Navorsing het ook getoon dat programmeringsbegrippe wel deur middel van *Scratch* aangeleer kan word, maar slegs indien geskikte onderrig–leerstrategieë gebruik word. Indien geskikte onderrig–leerstrategieë nie toegepas word nie, bestaan die moontlikheid dat swak programmeringsgewoontes met *Scratch* aangeleer kan word (sien 2.8.2.2).

Die volgende navorsingsvrae wat ondersoek gaan word, vloei direk uit voorafgaande voort, naamlik:

Hoe word *Scratch* tans deur onderwysers in die Noordwesprovinsie waar *Delphi* in graad 11 geïmplementeer is, onderrig en hoe het onderwysers die oorgang vanaf die onderrig van *Scratch* na die onderrig van *Delphi* ervaar?

Bostaande vrae is empiries ondersoek en in die volgende hoofstuk word die navorsingsontwerp en metodologie van die studie beskryf.

HOOFSTUK 3:

NAVORSINGSONTWERP EN METODOLOGIE

3.1 INLEIDING

In hoofstuk 2 is die aard van programmering en die programmeertale *Scratch* en *Delphi* bespreek. Daar is bevind dat daar verskeie ooreenstemmende programmeringsbeginsels in beide programmeertale voorkom (sien tabel 2.4). Dit behoort dus moontlik te wees om hierdie ooreenstemmende programmeringsbeginsels reeds met *Scratch* aan te leer. Malan en Leitner (2007:233) en Meerbaum-Salant *et al*. (2011:169) het egter bewys dat sekere programmeringsbeginsels pertinent in *Scratch* onderrig moet word vir leerders om ’n begrip daarvan te vorm en om goeie programmeringsgewoontes te ontwikkel (sien 2.8.2).

In hierdie hoofstuk word die navorsingsontwerp en die metodologie wat in hierdie navorsing gebruik is, bespreek.

3.2 DOEL VAN DIE NAVORSING

Die uiteindelike doel van die navorsing was om riglyne te gee vir die onderrig van *Scratch* ten einde die oorgang na *Delphi* te ondersteun. Met hierdie doel voor oë is die onderstaande navorsingsvrae tydens die empiriese navorsing ondersoek:

Hoe word *Scratch* tans in Noordwesprovinsie waar *Delphi* in graad 11 geïmplementeer is, onderrig? (navorsingsvraag 5)

Hoe het onderwysers die oorgang vanaf die onderrig van *Scratch* na die onderrig van *Delphi* ervaar? (navorsingsvraag 6)

3.3 PARADIGMA

Creswell (2009:6) beskryf die begrip *paradigma* as navorsers se algemene oriëntering teenoor die wêreld rondom hulle. Daar bestaan verskeie paradigmas waaruit opvoedkundige navorsing gedoen kan word, soos onder andere die

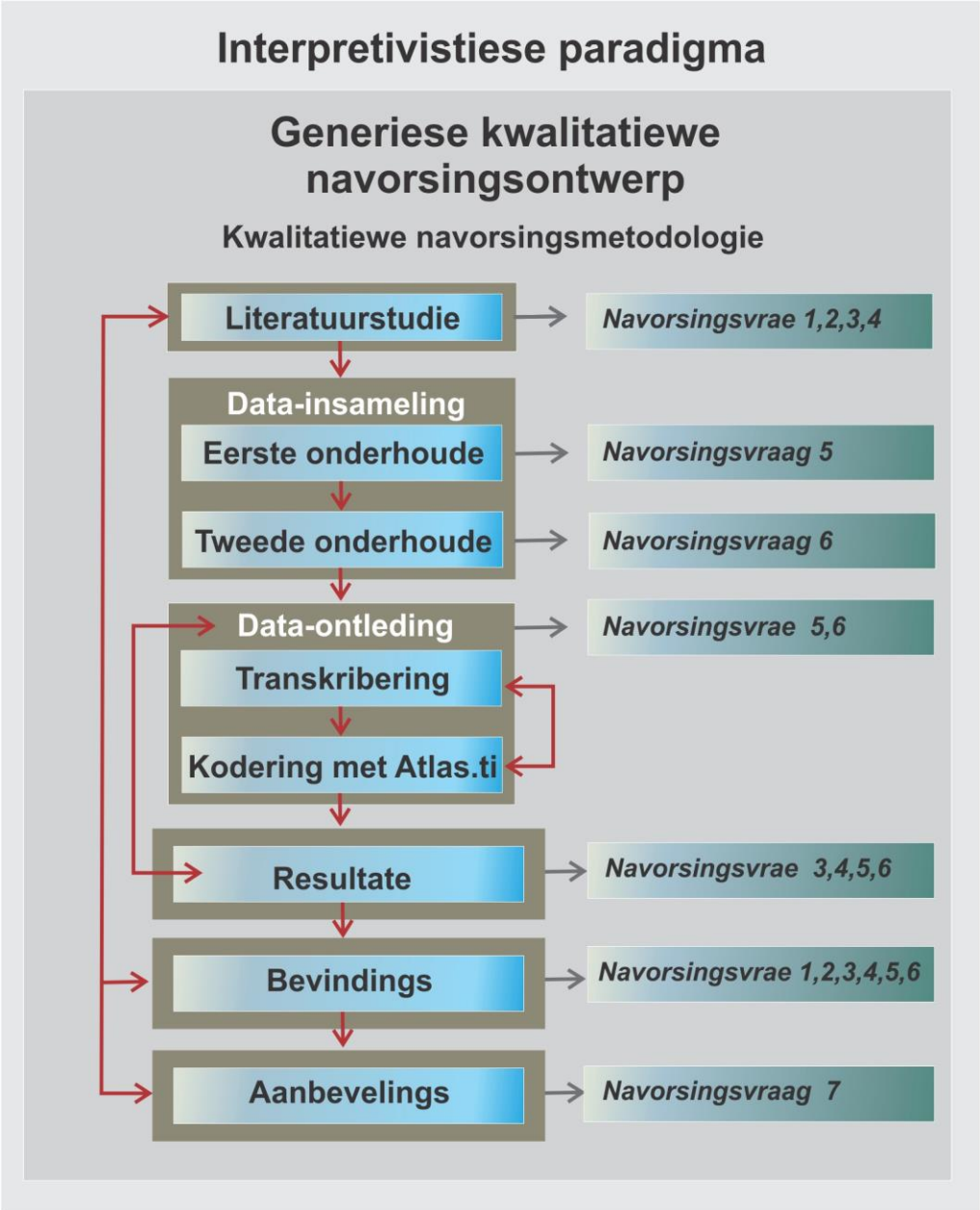
positivisme, konstruktivisme, interpretivisme en pragmatisme (Mackenzie & Knipe, 2006:194). Hierdie studie was in ’n interpretivistiese paradigma begrond, met ander woorde dit het gepoog om ’n diepere begrip te vorm van die ervaring van ’n individu (Schnelker, 2006:45). Volgens Nieuwenhuis (2007:59–60) plaas interpretivisme die betekenis wat deelnemers aan hulle ervarings heg op die voorgrond, en is die fokus op deelnemers se subjektiewe ervarings ten einde perspektief op ’n situasie te verkry.

Die navorser het op deelnemers se sienings (Creswell, 2009:8) van hul *Scratch*-onderrig en die oorgang na *Delphi* staatgemaak, sodat daar uiteindelik ’n diepere begrip verkry kan word van hulle *Scratch*-onderrig en hulle ervaring van die oorgang na *Delphi*. Kennis van die werklikheid is bepaal deur deelnemers se weergawes van hulle ervarings (Walsham, 1995:376). Volgens Van der Walt en Potgieter (2012:224) word deelnemers se opinies beïnvloed deur hulle lewensuitkyke, tradisies en kulture waarbinne hulle leef en werk. Daar kan gevolglik aanvaar word dat deelnemers se belewenisse oor die onderrig van *Scratch*, beïnvloed is deur tradisies en kulture waarbinne hulle leef en werk en hulle siening van wat waarheid en werklikheid is.

Die navorser as interpretivis kan self ook nie objektief teenoor die navorsing staan nie (Nieuwenhuis, 2007:54). Die navorser het haar egter daarvan weerhou om persoonlike voorkeure tydens onderhoude uit te spreek en het gepoog om resultate uit data so getrou moontlik weer te gee

3.4 NAVORSINGSMETODOLOGIE

Binne die interpretivistiese paradigma is ’n kwalitatiewe navorsingsmetodologie vir hierdie studie geselekteer. Volgens Olivier (2009:111) word ’n kwalitatiewe navorsingsmetodologie gebruik wanneer ’n diepere begrip van ’n bepaalde situasie benodig word, en dit word deur middel van diepgaande sienings van deelnemers verkry. Omdat die fokus van die huidige studie was om deelnemers se ervaring oor die onderrig van *Scratch* en die oorgang na *Delphi* grondig te ondersoek, is kwalitatiewe navorsing as ’n geskikte navorsingsmetodologie beskou. Figuur 3.1 gee ’n uiteensetting van die navorsingsontwerp en –metodologie en hoe elke navorsingsvraag aangespreek is.



Figuur 3.1: Navorsingsontwerp en navorsingsmetodologie

Vervolgens word die navorsingsontwerp ten opsigte van die populasie en steekproef, data-insameling, etiese aspekte en data-ontleding, rol van die navorser, verklaring van terme wat in die navorsing gebruik word en verifiëring van resultate bespreek.

3.5 NAVORSINGSONTWERP

'n Navorsingsontwerp word met inagneming van 'n bepaalde paradigma gekies (Creswell, 2009:11). 'n Generiese kwalitatiewe navorsingsontwerp (Merriam, 1998:11) is vir die studie gebruik, aangesien die doel van die navorsing was om

bepaalde fenomene, naamlik die onderrig van *Scratch* en die oorgang na *Delphi*, te begryp, om uiteindelik hierdie bevindings sowel as vorige navorsing op hierdie terrein te gebruik en aanbevelings vir die onderrig van *Scratch* te maak. Vervolgens word aspekte met betrekking tot die navorsingsontwerp, naamlik die seleksie van deelnemers, data-insamelingstegnieke en die metodes wat vir data-analise gebruik is, bespreek.

3.5.1 Populasie en steekproef

Die teikenpopulasie van die studie was al die IT-onderwysers in Noordwesprovinsie (n=24). ’n Homogene, ewekansige steekproefmetode is gebruik om tien deelnemers vir die studie te selekteer. Volgens Creswell (2008:216) word homogene steekproefneming gebruik indien deelnemers ’n sekere gemeenskaplike eienskap moet besit. In hierdie studie was die gemeenskaplike eienskap dat die deelnemers in 2012 graad 10 IT-leerders en in 2013 graad 11 IT-leerders onderrig het.

’n Genommerde lys van IT-onderwysers in Noordwesprovinsie is vanaf die Noordwes departement van onderwys verkry. Die name was in geen spesifieke volgorde nie. Die navorser het ’n *Scratch*-program geskryf om ’n ewekansige trekking van nommers te doen. Onderwysers is toegeken volgens die nommers wat getrek is. Indien die onderwyser *Scratch* in 2012 en *Delphi* in 2013 onderrig het, is die onderwyser as deelnemer vir die studie geïdentifiseer. Die eerste tien onderwysers wat aan die vereistes van die homogene steekproef voldoen het, is as deelnemers geselekteer.

3.5.2 Data-insameling

Semi-gestruktureerde onderhoude is as metode van data-insameling gekies. Sodoende kon spesifieke sowel as gedetailleerde inligting met betrekking tot die onderrig van *Delphi* en *Scratch* verkry word. Daar is besluit om persoonlike onderhoude met deelnemers te voer, sodat insig oor elke deelnemer se persoonlike ervarings verkry kon word (Olivier, 2009:111) en deelnemers nie deur die opinie van ander deelnemers beïnvloed sou word nie (Creswell, 2008:225–226). Met onderhoude as metode van data-insameling was die doel om ’n ryk verskeidenheid data te verkry (Olivier, 2009:11) sodat die navorser onderrigriglyne kon saamstel om die oorgang vanaf *Scratch* na *Delphi* te ondersteun.

Volgens Creswell (2008:443) word data-saturasie verkry indien geen nuwe temas uit onderhoude geïdentifiseer word nie. Daar is gevolglik beplan dat onderhoude gevoer sou word totdat data-saturasie voorkom. Nadat onderhoude met ses deelnemers gevoer is, het data-saturasie voorgekom, maar onderhoude is egter met nog twee verdere deelnemers gevoer om te verseker dat genoeg data ingesamel word.

Dit was nodig om twee stelle individuele onderhoude met elke deelnemer te voer. In tabel 3.1 word die tydlyn van die onderhoude met deelnemers weergegee. Die eerste onderhoude is aan die begin van 2013 met IT-onderwysers gevoer, om insig te verkry in hoe hulle *Scratch* in 2012 aan graad 10-leerders onderrig het. Ná verloop van ongeveer ses maande se *Delphi*-onderrig aan graad 11-leerders is 'n tweede onderhoud met elke deelnemer gevoer. Die doel met die tweede onderhoude was om die deelnemers in staat te stel om hulle ervarings ten opsigte van die oorgang vanaf *Scratch* na *Delphi* met die navorser te deel, sodat die navorser 'n deeglike begrip daarvan kon kry (Babbie & Mouton, 2001:33,309).

Tabel 3.1: Tydlyn van onderhoude

<table>
<tr><th>JAAR</th><th colspan="4">PROGRAMMEERTAAL AANGEBIED IN DIE VAK IT</th></tr>
<tr><td>2012</td><td colspan="4">← Scratch: graad 10 →</td></tr>
<tr><td rowspan="3">2013</td><td colspan="4">← Delphi : graad 11 →</td></tr>
<tr><td>Jan–Mar</td><td>Apr–Jun</td><td>Jul–Aug</td><td>Sep–Des</td></tr>
<tr><td>Eerste onderhoude
Hoe Scratch in 2012 aan graad 10-leerders onderrig is.</td><td>Onderrig in Delphi aan graad 11-leerders.</td><td>Tweede onderhoude
Ervarings tydens die oorgang van Scratch na Delphi.</td><td>Onderrig in Delphi aan graad 11-leerders.</td></tr>
</table>

Uit die literatuur is bepaalde aspekte oor die aard van programmering (sien 2.3), die aard van *Scratch* (sien 2.4.2) en die aard van *Delphi* (sien 2.5.2) vooraf geïdentifiseer. Hierdie aspekte is ingesluit by die vrae wat in die semi-gestruktureerde onderhoude aan deelnemers gestel is. Volgens Nieuwenhuis (2007:87) word semi-gestruktureerde onderhoude gebruik om data te bevestig wat uit ander bronne bekom is, deurdat deelnemers voorafopgestelde vrae moet beantwoord en die navorser deelnemers verder kan uitvra indien hulle antwoorde nie

duidelik is nie. Dieselfde formaat as vir die eerste onderhoude is vir die tweede stel onderhoude gevolg. Die vrae wat tydens die eerste en tweede onderhoude aan deelnemers gestel is, sowel as die motivering daarvan, word in tabel 3.2 en tabel 3.3 bespreek.

Tabel 3.2: Eerste onderhoud

VRAAG	MOTIVERING
Verduidelik wat u doel was met die onderrig van *Scratch*.	*Scratch* kan om verskeie redes onderrig word, naamlik om programmeringsbeginsels aan te leer (sien 2.4.1), om leerders se nuuskierigheid oor programmering te prikkel, om leerders se logiese denke te ontwikkel en om 'n pret-omgewing te skep om stories te vertel en speletjies te speel. Die navorser wou bepaal wat elke deelnemer se doelwit was met die onderrig van *Scratch*.
Verduidelik watter onderrig–leerstrategie u gebruik het om *Scratch* te onderrig.	In hoofstuk 2 is bevind dat onderrig–leerstrategie 'n belangrike rol speel by die sukses van onderrig (sien 2.7.1 en 2.8.1). Die onderrig–leerstrategie wat deelnemers toegepas het, moes dus bepaal word.
Verduidelik watter spesifieke programmeringsbeginsels/-begrippe u met *Scratch* onderrig het en hoe u dit gedoen het.	By hierdie vraag was die doel om te bepaal of die deelnemer programmeringsbeginsels/-begrippe bewustelik onderrig het en om ondersoek in te stel hoe dit gedoen is.
Hoe lank het u *Scratch* in graad 10 onderrig voordat u met *Delphi* begin het?	By hierdie vraag was die doel om te bepaal of die deelnemers die totale tyd wat vir *Scratch*-onderrig toegeken word, daaraan bestee het.

Tabel 3.3: Tweede onderhoud

VRAAG	MOTIVERING
Hoe beleef graad 11-leerders die oorgang van *Scratch* na *Delphi*?	Hier was die doel om die deelnemer die geleentheid te gee om oor die graad 11-leerders se ervaring van die oorgang na *Delphi* te reflekteer.
Watter afdelings van die werk of begrippe van programmering in *Delphi* het u al met die leerders behandel?	Met hierdie vraag het die navorser 'n drieledige doel gehad: • om deelnemers die geleentheid te gee om te reflekteer oor die werk wat reeds behandel is; • om te bepaal watter programmeringsbegrippe reeds in *Delphi* onderrig is; en • om die tempo van onderrig te ondersoek. Daar is gehoop dat bogenoemde moontlik 'n aanduiding sou kon gee of oordrag geskied het en of programmeringsbeginsels/-begrippe van vooraf aangeleer moes word.
Hoe vergelyk leerders se begrip van *Delphi*-programmering met die vorige jare toe *Scratch* nie in graad 10 onderrig is nie?	Die navorser wou bepaal of *Scratch*-onderrig bygedra het tot 'n begrip van programmering in *Delphi* en wat die mate van oordrag was.
Waarmee sukkel die leerders in *Delphi* en is daar begrippe wat hulle maklik vind?	Die doel was om vas te stel of oordrag van begrippe plaasgevind het en waar oordrag van begrippe nie plaasgevind het nie.
Het u tendense waargeneem waar leerders heeltyd dieselfde foute maak of swak programmeringstegnieke toepas?	Aangesien daar volgens hoofstuk 2 in die literatuur beskryf is dat bepaalde tendense by die onderrig van *Scratch* waargeneem is (sien 2.8.2), wat algemene programmeringstegnieke beïnvloed, wou die navorser bepaal of tendense ook voorgekom en na *Delphi* oorgedra is.
Kan u enige opmerkings maak oor die invloed van *Scratch* op die aanleer van *Delphi*?	Die navorser wou bepaal of deelnemers voel dat *Scratch* waarde het vir die aanleer van programmeringsbegrippe ten einde *Delphi* te ondersteun.

In die volgende gedeelte word enkele aspekte met betrekking tot die omstandighede om die voer van onderhoude heen bespreek.

Die onderhoude is op geskikte tye, wat nie met onderrigverpligtinge van deelnemers ingemeng het nie, gevoer. Aangesien die onderhoude elektronies opgeneem is, moes ’n geskikte, stil omgewing gebruik word. Twee opneemtoestelle, naamlik ’n slimfoon en digitale opneemtoestel is gelyktydig gebruik, om te verseker dat data nie verlore sou gaan indien ’n toestel onklaar sou raak nie. Die toestelle is vooraf getoets vir reikwydte van ontvangs, aangesien deelnemers soms sag en onduidelik praat (Creswell, 2008:229). Die onderhoudvoerder het ook ’n onderhoudsprotokol saamgeneem (sien addendums D en E), met die onderhoudsvrae vooraf uitgetik en ruimte vir aantekeninge. Die doel van sodanige vorm is om die onderhoudvoerder te herinner aan die vrae wat gestel moet word en om aantekeninge oor emosie, gesigsuitdrukkings en ander sake te maak wat nie op die opname vasgelê kan word nie. So ver moontlik is gepoog om by die voorafopgestelde vrae te hou, maar ruimte is toegelaat vir buigsaamheid (Creswell, 2008:229) ten einde verdere uitklaring van deelnemers se antwoorde te verkry.

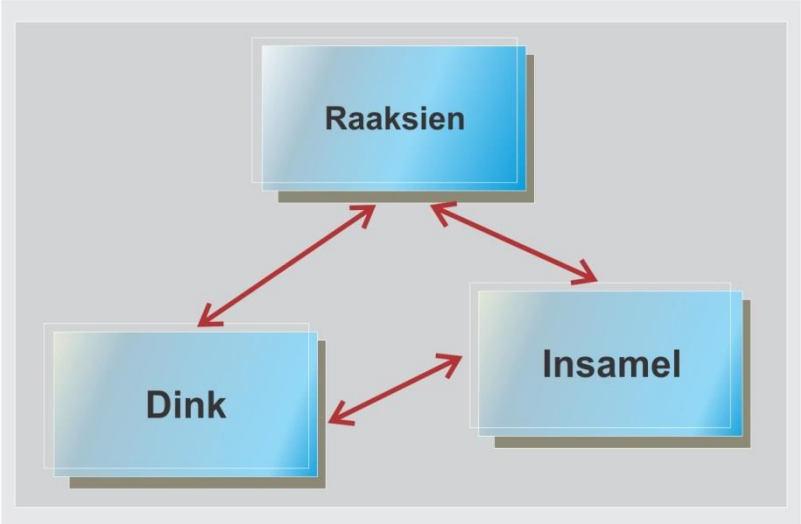
3.5.3 Etiese aspekte

Die doel van die onderhoude is vooraf aan deelnemers verduidelik en deelnemers is ingelig dat hulle onder geen verpligting staan om aan die navorsing deel te neem nie en dat hulle te eniger tyd kon onttrek. Aan die einde van elke onderhoud is deelnemers vir hulle deelname bedank en weer eens van die vertroulikheid van die data verseker. Deelnemers se name is nêrens in die transkripsies opgeteken nie en anonimiteit is deur die navorser wat self die onderhoude gevoer en die data ontleed het, verseker. Alhoewel deelnemers sekere demografiese data bekend moes maak, soos geslag, kwalifikasie en aantal jare ervaring in IT-onderrig, is geen sensitiewe of vertroulike inligting van deelnemers versoek nie. Indien vertroulike aspekte deur deelnemers in onderhoude genoem of bespreek is, is sodanige inligting nie bekend gemaak nie. Die data word op ’n veilige plek gestoor en word deur ’n wagwoord beskerm.

Soos vermeld in hoofstuk 1 (sien 1.5), is etiese toestemming van die NWU, sowel as die Noordwes Departement van Onderwys en die hoofde van deelnemende skole verkry. Ingeligte toestemmingsvorms is deur skoolhoofde en deelnemers voltooi (sien addendums B en C).

3.5.4 Data-ontleding

Volgens Gibbs (2007:38) is kodering ’n manier om teks te indekseer of te kategoriseer ten einde ’n raamwerk van temas te verkry. Friese (2012:93) se kwalitatiewe data-analise proses van raaksien, insamel en dink (sien figuur 3.2) is gevolg om kodes aan data toe te ken. Die RID-model skryf nie ’n spesifieke volgorde van handelinge voor nie, maar behels dat die navorser bepaalde aspekte raaksien, daaroor nadink, dit insamel en gedurig, soos nodig, handelinge afwissel.



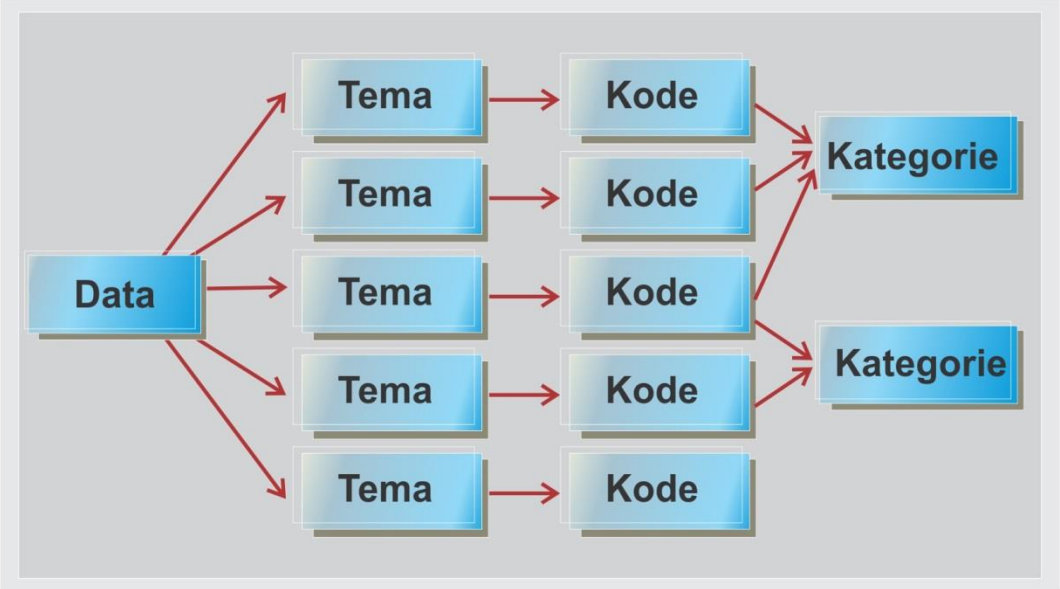
Figuur 3.2: Die RID-model (Friese, 2012:93)

Ná elke onderhoud is die data onmiddellik getranskribeer en ontleed. Sodoende kon telkens bepaal word of data-saturasie verkry is (sien 3.5.2). Data-ontleding is met behulp van die Atlas.ti-rekenaarprogram gedoen wat die navorser in staat gestel het transkripsies van die onderhoude te kodeer en die kodes te ontleed. ’n Data-gedrewe werkswyse van kodering (Gibbs, 2007:45) is toegepas, waar kodes induktief (Friese, 2012:93) ontwikkel is, soos wat transkripsies deurgelees is. Tydens hierdie eerste fase van kodering is baie kodes verkry waarvan die meeste ’n lae frekwensie gehad het. In ooreenstemming met Friese (2012:108) se riglyne is die RID-model herhaaldelik toegepas om kodes te konseptualiseer en saam te voeg om kategorieë (sien figuur 3.3) te vorm (Friese, 2012:108). ’n Navorsingsdagboek (Friese, 2012:137) is in Atlas.ti opgestel om die proses van data-ontleding te beskryf.

In 3.5.5 word die terme *tema*, *kode* en *kategorie* belig.

3.5.5 Verklaring van terme

Die terme *tema*, *kode* en *kategorie* word in die res van hierdie navorsing gebruik. In figuur 3.3 word aangedui wat die verband tussen die terme is en hoe die data-ontleding gedoen is. Die woord *tema* verwys na ’n gedagte wat herhaaldelik in ’n transkripsie voorkom (Gibbs, 2007:152). Aan elke tema word ’n bepaalde beskrywing gekoppel, wat ’n *kode* genoem word (Gibbs, 2007:148). Kodes wat aan mekaar verwant is, word saamgevoeg in *kategorieë*. Kodes behoort nie eksklusief aan ’n bepaalde kategorie nie, maar kan aan verskeie kategorieë toegeken word. Dit is ook moontlik dat ’n kode nie aan ’n bepaalde kategorie toegeken word nie.



Figuur 3.3: Skematiese voorstelling van terme by data-ontleding

3.5.6 Rol van die navorser

Die navorser was deurgaans alleen betrokke by die insameling en ontleding van die data (sien 3.5.2). Alhoewel die navorser gepoog het om so objektief moontlik te wees, beweer Olivier (2009:112) dat die navorser nooit ten volle objektief en afsydig in die situasie is nie, omdat ’n navorser ook altyd sy/haar eie siening van die werklikheid het. Daar is egter gestreef na objektiwiteit deur die onderhoudsvrae noukeurig te kies en deelnemers nie te lei om bepaalde antwoorde te gee nie. Tydens onderhoudvoering kan deelnemers dalk misleidende inligting verskaf, dit wat

hulle dink die navorser wil hoor, of die teenwoordigheid van die navorser kan die manier waarop 'n deelnemer reageer, beïnvloed (Creswell, 2008:226). Die navorser het dit probeer te bowe kom deur so min moontlik op deelnemers se terugvoer te reageer en deelnemers eerder aan te moedig om te praat. Na aanleiding van Creswell (2008:239) se aanbevelings het die navorser gepoog om tydens haar reaksie op deelnemers se terugvoer objektief te bly en nie haar mening oor die onderrig van *Scratch* en *Delphi* te lug nie. Opvolgvrae is gevra indien nodig ten einde 'n beter begrip van deelnemers se terugvoer en ervaring te verkry.

3.5.7 Verifiëring van resultate

Kwalitatiewe navorsing maak van taal gebruik, en verskeie interpretasies van data is om hierdie rede moontlik (Olivier, 2009:112; Ponterotto, 2005:130). Ponterotto (2005:130) beweer dat 'n interpretivistiese navorser nie data-analise ekstern hoef te verifieer nie, aangesien 'n ander navorser heel moontlik die data anders sal interpreteer en gevolglik ander temas uit die data sal identifiseer. Volgens Ponterotto (2005:130) is die leser van die navorsingsresultate die persoon wat 'n oordeel oor die geldigheid daarvan moet vel. Maree en Van der Westhuizen (2007:38) sluit hierby aan deur te meld dat die geldigheid ("validity") van bevindings in kwalitatiewe studies bepaal word deur die mate van die gemeenskaplike betekenis van bevindings vir die navorser en die deelnemer. Creswell (2008:266) en Morse *et al*. (2002:13) beweer egter dat dit belangrik is dat die navorser deur die hele proses van data-ontleding die akkuraatheid van kwalitatiewe bevindings verseker en dit verifieer.

Morse *et al*. (2002:16) noem dat dit te laat is om resultate ekstern te verifieer en geloofwaardigheid te bepaal indien die navorsing reeds afgehandel is. Hulle beweer dat die terme *betroubaarheid* ("reliability") en *geldigheid* by kwalitatiewe verifiëring van navorsing ingestel moet word (sien 1.4.5). Betroubaarheid en geldigheid kan volgens hulle bereik word indien die navorser deurgaans alle handelinge verifieer en gedurig kreatief en met insig en openheid reageer.

In hierdie navorsing is gepoog om betroubaarheid en geldigheid te verkry deur die benadering van Morse *et al*. (2002:16), soos bo bespreek, te volg. Daar is herhaaldelik na klankopnames van onderhoude geluister om te verseker dat die transkripsies korrek is en dat die navorser die ervarings van deelnemers reg

geïnterpreteer het. Die RID-model (sien figuur 3.2) is gevolg om data te analiseer en het ook bygedra dat die navorser herhaaldelik oor die data besin het en die koderingsproses herhaaldelik uitgevoer het om seker te maak dat alle aspekte raakgesien en in die kodes verreken is. Die toekenning van kodes en die samevoeging daarvan in kategorieë is gevolglik telkens nagegaan en soms verander en verfyn, soos wat die transkripsies herhaaldelik deurgelees is.

In tabel 3.4 word verwys na sommige strategieë om betroubaarheid en geldigheid van kwalitatiewe navorsing te bepaal, soos deur Morse *et al*. (2002:18–19) voorgestel. Daar word ook aangedui hoe dit in hierdie navorsing toegepas is.

Tabel 3.4: Strategieë vir verifiëring van die navorsingsproses

STRATEGIE	TOEPASSING IN DIE NAVORSING
Verseker metodologiese samehangendheid.	Die navorser het deurgaans gestreef dat die navorsingsontwerp en alle aspekte daarvan by ’n kwalitatiewe navorsingsmetodologie pas.
Die steekproef moet toepaslik wees.	Geskikte deelnemers is gekies (sien 3.5.1). Data-saturasie was ’n aanduiding dat data-insameling volledig was (Morse *et al*., 2002:12).
Insameling van data en data-analise moet gelyktydig geskied.	Soos wat die data ingesamel is, is dit ontleed (sien 3.5.2).

3.6 SAMEVATTING

In hierdie hoofstuk is die paradigma, navorsingsontwerp en navorsingsmetodologie van die navorsing bespreek, soos opsommend grafies in figuur 3.1 aangetoon. Die studie was in ’n interpretivistiese paradigma gefundeer en ’n generiese kwalitatiewe navorsingsontwerp is gevolg. Die populasie vir die studie was onderwysers in Noordwesprovinsie wat IT as vak vir graad 10- en graad 11-leerders aanbied en agt onderwysers het uiteindelik aan die navorsing deelgeneem. Data-insameling het deur middel van individuele semi-gestruktureerde onderhoude geskied, en twee stelle individuele onderhoude is met elke deelnemer gevoer. As gevolg van die navorsingsvrae wat beantwoord moes word en die aard van die data wat ingesamel moes word, was die onderhoude tydsgebonde. Onderhoude is aan die begin van 2013 en gedurende Julie 2013 gevoer. Ná elke onderhoud is ontleding van die data

gedoen om te bepaal of data-saturasie verkry is. Die navorser was deurgaans betrokke by die insameling en ontleding van data en het sover moontlik gepoog om alle etiese aspekte na te kom, objektief te wees en nie haar eie opinies aan deelnemers bekend te maak nie. In die volgende hoofstuk word die bevindings en resultate van data-ontleding gerapporteer en bespreek.

HOOFSTUK 4:

RAPPORTERING EN BESPREKING VAN RESULTATE

4.1 INLEIDING

In hoofstuk 2 is die aard van *Scratch* en *Delphi* as programmeertale ondersoek. Daar is onder andere gekyk na die aard van die programmeertale, hoe die twee programmeertale onderrig behoort te word en of daar gemeenskaplike begrippe in beide programmeertale voorkom. 'n Empiriese studie is uitgevoer om die onderstaande navorsingsvrae te beantwoord, naamlik:

- Hoe word *Scratch* tans in die Noordwesprovinsie waar *Delphi* in graad 11 geïmplementeer is, onderrig? (navorsingsvraag 5)
- Hoe het onderwysers die oorgang vanaf die onderrig van *Scratch* na die onderrig van *Delphi* ervaar? (navorsingsvraag 6)

In hierdie hoofstuk word die bevindings van die empiriese studie, met die oog op bestaande twee navorsingsvrae gerapporteer. Verskeie temas het tydens die analise van data na vore gekom. Hierdie temas is gekodeer na kodes en in kategorieë ingedeel soos dit konseptueel nodig was.

Resultate met betrekking tot die navorsingsvrae, naamlik hoe *Scratch* tans deur onderwysers in die Noordwesprovinsie, waar *Delphi* in graad 11 geïmplementeer word, onderrig word, en hoe onderwysers die oorgang na *Delphi* ervaar het, word in die onderstaande afdelings gerapporteer.

4.2 ERVARING MET DIE ONDERRIG VAN *SCRATCH*

Verskeie kategorieë, waaronder onderwysers se onsekerheid en kwessies rakende handboeke, die onderrig van probleemoplossing en algoritme-ontwerp, die onderrig van ander programmeringsbeginsels en -begrippe, onderrig–leerstrategieë en metodes wat onderwysers toepas en die tyd wat aan *Scratch*-onderrig bestee word, is ten opsigte van die onderwysers se ervaring met die onderrig van *Scratch* uit die

onderhoude geïdentifiseer. Die bevindings word vervolgens ooreenkomstig die geïdentifiseerde kategorieë gerapporteer.

4.2.1 Onsekerheid van onderwysers en kwessies rakende handboeke

Kodes in hierdie kategorie hou verband met onderwysers se onsekerheid en aspekte rakende handboeke. Onderwysers was onseker oor hoe om *Scratch* te onderrig. Boonop was handboeke nie aanvanklik beskikbaar nie, en dit het bygedra tot die onsekerheid ten opsigte van die onderrig van *Scratch*. Deelnemers se response word so ver moontlik woordeliks verskaf.

> *Verlede jaar, toe ek [met Scratch-onderrig] begin het, het ek half rondgeval (P2).*
>
> *Ek was ook maar nie heeltemal seker van Scratch nie en ek dink ek het dit dalk te volledig gedoen (P4).*

'n Rede vir die onsekerheid wat deur verskeie deelnemers aangevoer is, was dat handboeke eers laat in die jaar beskikbaar was.

> *In Julie, nee Junie het ek eers handboeke gekry. So ek het my eie data uit die lug getrek en werk gedoen (P3).*

Onderwysers het oor die algemeen erken dat hulle baie handboekgerig werk en daarop steun vir riglyne betreffende hulle onderrig. Hulle maak staat op die volgorde en diepte van werk soos wat dit in die handboek behandel word.

> *Aanvanklik het ek die handboek slaafs nagevolg (P2).*
>
> *Ek is baie handboekgerig; ek volg die voorbeelde uit die handboek (P4).*
>
> *Ek het hulle elke voorbeeld in die boek laat doen. Ek het nie keuses gegee nie; ek het nie gesê doen byvoorbeeld nommmers 1, 7 en 9 nie. Ek het dat hulle elke voorbeeld in die boek doen (P4).*

Slegs deelnemer P8 het aanvanklik aangedui dat handboeke nie slaafs gevolg word nie.

> *Ek is nie mal om aktiwiteite uit 'n handboek te gee nie, want ek dink dit is net belangrik om 'n handboek as beginpunt te gebruik. Die handboek se aktiwiteite is half vervelig vir leerders en jy forseer die kind om 'n spesifieke iets te doen (P8).*

Bogenoemde opmerking was egter in kontras met deelnemer P4 se opmerking dat leerders elke voorbeeld in die handboek doen.

Deelnemer P8 het egter later, in teenstelling met haar vorige stelling genoem dat sy tog soms handboekgerig werk.

> *Toe ons vloeidiagramme behandel het, was ek handboekvas gewees (P8).*

Voordat handboeke beskikbaar was, het onderwysers *Scratch* dus aanvanklik lukraak aangebied, gelei deur die *Scratch*-omgewing self en nie volgens temas soos in die KABV-dokument uiteengesit nie.

> *Ek het* Scratch *gevat en gesê kom ons begin by die eerste groep kode, dan die tweede, en so het ek aangegaan en dit saam met hulle uitgewerk. Die kinders het dit verstaan, maar in die handboek was dit beter uiteengesit (P3).*

Een onderwyser het wel aangedui dat sy riglyne in die KABV-dokument gevolg het.

> *Programmering het 'n wye omvang, maar ek kyk na die kurrikulum om die diepte van programmering wat van leerders verwag word, te bepaal (P5).*

Dieselfde onderwyser het egter later vertel dat alhoewel sy bewus was van die inhoud van die kurrikulum, sy haar deur die leerders laat lei het.

> *Met Scratch het ek net my leerders gevolg (P5).*

Ten spyte daarvan dat onderwysers onseker was oor hoe om *Scratch* te onderrig, het leerders intussen *Scratch* aangeleer deur ander bronne te gebruik en met *Scratch* geëksperimenteer. Onderwysers het gevolglik minderwaardig teenoor leerders gevoel, aangesien leerders meer vertroud was met *Scratch* as wat hulle was.

> *Die kinders het alreeds Desember met Scratch begin speel, so vir my as onderwyser om voor hulle te kom staan het, was nogal 'n probleem, want die kinders voel dat hulle my vyf tree voor is. Daar is nie handboeke op die stadium nie en die oefeninge wat ek moes uitdink het nie so lekker by hulle posgevat nie (P6).*

Die kwessie van handboeke het herhaaldelik in verskeie gesprekke met deelnemers voorgekom, sonder dat die navorser in die vrae wat aan deelnemers gestel is, daarna verwys het. Alhoewel onderwysers vooraf opleiding oor die inhoud van die KABV ontvang het, is hulle slegs aan *Scratch* bekend gestel en het nie opleiding ontvang oor moontlike onderrig–leerstrategieë wat met *Scratch* gevolg kon word nie. Gevolglik was hulle onseker oor die onderrig van *Scratch*. Die onsekerheid van onderwysers oor die onderrig van *Scratch* stem ooreen met Kölling (2008:99) en Meerbaum-Salant *et al.* (2013:171) se bevindings dat 'n nuwe

programmeringsomgewing onderwysers onseker laat voel en dat korrekte onderrigpraktyke nie noodwendig aanvanklik gevolg sal word indien hulle onderrigmetodes op hulle eie moet ontdek nie (sien 2.8.1).

Uit bostaande bespreking word die gevolgtrekking gemaak dat onderwysers, veral weens hulle onsekerheid, op handboeke as riglyn vir hulle onderrig staatmaak en dat dit hulle eerste naslaanbron is.

Samevattend kan dus gesê word dat onderwysers onseker was oor hoe Scratch onderrig moes word, sowel as die inhoud wat onderrig moes word. Daar was aanvanklik nie handboeke beskikbaar nie en dit het onderrig verder bemoeilik. *Scratch*-onderrig was gevolglik gebaseer op onderwysers se eie interpretasie van die *Scratch*-programmeringsomgewing en nie noodwendig op die volgorde wat in die KABV aangedui word nie. Alhoewel die KABV-dokument wel riglyne en wenke oor die onderrig van *Scratch* bevat, is die gevolgtrekking ook gemaak dat onderwysers eerder op handboeke staatgemaak het vir die interpretasie daarvan. Die laat aflewering van handboeke het daartoe bygedra dat die deelnemende onderwysers aanvanklik rondgeval het met *Scratch*-onderrig en waardevolle tyd vir die onderrig van spesifieke programmeringsbeginsels en -begrippe, soos deur die KABV-dokument aangedui, verloor het.

In die volgende afdeling word die onderwysers se ervarings rakende die onderrig van programmeringsbeginsels gerapporteer. Kodes met betrekking tot hierdie aspek is in kategorieë gegroepeer en vier verskillende kategorieë is geïdentifiseer – naamlik die onderrig van probleemoplossing en algoritme-ontwerp (sien figuur 2.9), die onderrig van ander programmeringsbegrippe, onderrig–leerstrategieë en metodes vir die onderrig van *Scratch* en die tyd wat aan *Scratch*-onderrig bestee moet word.

4.2.2 Onderrig van probleemoplossing en algoritme-ontwerp

Verskeie kodes wat met die onderrig van probleemoplossing en algoritme-ontwerp verband hou, het tydens die data-insameling na vore gekom – naamlik die tyd wat aan algoritme-ontwerp bestee moet word, beplanning, en waar algoritme-ontwerp by die onderrig van programmering in *Scratch* moet inpas. Dit was duidelik dat onderwysers die bevordering van logiese denke en die aanleer van probleemoplossing as die belangrikste aspekte van *Scratch*-onderrig beskou. Die

meerderheid onderwysers was eenstemmig dat algoritme-ontwerp en probleemoplossing belangrike vaardighede was om te onderrig en daarom het meeste onderwysers daarop aangedring dat ’n algoritme of vloeidiagram ontwerp moes word voordat die oplossings in *Scratch* geprogrammeer kan word.

> *Ek dink dit gaan alles oor logiese denke, probleemoplossing. Eerstens moet hulle enige probleem kan oplos deur middel van logiese denke. Dit is, dink ek, die belangrikste. Ek het báie tyd spandeer aan vloeidiagramme en algoritmes (P1).*
>
> *Ek dink ek het baie tyd spandeer aan logika en dalk te laat met Scratch begin (P1).*
>
> *Probleemoplossing is die belangrikste (P5).*
>
> *Met elke program wat my kinders skryf as hulle ’n opdrag kry, moet hulle eers ’n algoritme of ’n vloeidiagram ontwerp (P3).*
>
> *Dit is ’n proses van analiseer die vraag, ontwerp ’n toevoer–verwerking–afvoer-diagram en skryf dan die pseudokode. Daarna programmeer hulle. Dit moet laaste gebeur (P5).*
>
> *In die begin doen ons ongeveer ’n week lank algoritmes, daarna bly ek daarop fokus (P3).*
>
> *Die begin van die jaar, ja die begin van die jaar, begin ek met algoritmes (P4).*
>
> *Die eerste ding is maar om heeltemal die rekenaars af te sit. Daardie eerste hoofstuk is goed ontwerp in Scratch handboeke om net vloeidiagramme en die vloei van instruksies te doen (P6).*

Deelnemer P2 het aangevoer dat leerders nie daarvan hou om algoritmes te ontwerp nie, maar het aanbeveel dat leerders dit wel moet doen, aangesien dit hulle help om programme te beplan.

> *Hulle is nie gelukkig oor die papierwerk nie, maar hulle leer baie. Hulle leer dat hulle meer moet beplan (P2).*

Behalwe beplanning, help die ontwerp van algoritmes, volgens deelnemer P6, ook om deursettingsvermoë by leerders te kweek.

> *Hulle het daardie gevoel gekry van, komaan, ek moet nou dit doen (P6).*

Die eerste hoofstuk in die handboek handel oor algoritme-ontwerp en gevolglik het die deelnemende onderwysers gerapporteer dat hulle eers ’n tyd lank algoritmes behandel het. Die KABV (DBE, 2011:21) stel ook hierdie werkswyse voor. Volgens deelnemer P6 is ’n verdere voordeel van hierdie werkswyse dat dit klasdissipline bevorder en leerders op programmering laat fokus.

> *Die kinders het baie vloeidiagramme uitgewerk en dit het baie gehelp met klasdissipline, want mens sukkel verskriklik in programmering as jy agterkom almal sukkel met een probleem, om almal se aandag weer op die bord te kry. My onderrigstrategie is om aan die begin van die jaar regtig vir die kinders te laat fokus waarheen ons op pad is (P6).*

Daar was egter onderwysers wat meen dat algoritmes nie so belangrik is nie. Twee onderwysers was van die opinie dat algoritmes nie met die skryf van elke program afgedwing moet word nie, maar dat leerders dit tog gereeld moet doen. Daar is ook gemeld dat algoritmes vir toetsing gebruik kan word.

> *Hulle moet die algoritme toets met die toetsdata. As ek nou klaar verduidelik het, dan vra ek hulle om vinnig 'n plan neer te skryf hoe hulle dit gaan doen. Daar is nie altyd tyd om 'n algoritme te doen nie, maar mens kan dit so twee keer 'n week inpas (P4).*

Deelnemer P1 het nie in 2012 vereis dat 'n algoritme voor elke *Scratch*-program ontwerp moet word nie, maar was van voorneme om dit in die toekoms so te doen.

> *Ek het nie verlede jaar nie maar ek dink ek gaan dit hierdie jaar so doen. Dat hulle eers hulle denke oopkry en dink (P1).*

Deelnemer P7 was die enigste onderwyser wat nie algoritme-ontwerp in 2012 toegepas het nie en steeds nie dink dis belangrik vir *Scratch*-onderrig nie. Deelnemer P8 het ook nie algoritme-ontwerp in 2012 toegepas nie, maar erken dat dit problematies was en sal dit nie weer in die toekoms so doen nie.

> *Ek het hierdie jaar [2013,* Scratch*] met algoritmes begin, waar ek laasjaar [2012,* Scratch*] eers net begin programmeer het en dit was 'n fout, want as mens met algoritmes begin, is dit vir hulle baie vervelig en hulle weet nie hoekom nie en hulle weet nie waarom dit gedoen word nie. Die graad 10-handboeke begin met algoritmes en dit werk glad nie so nie. Ek sal persoonlik eers programmering vir hulle probeer leer, ek dink die speel-deel is vir hulle lekker (P7).*

> *Ek het die vorige jaar [2012] nie algoritmes gedoen nie, net weggeval met Scratch. Eers algoritmes en dan Scratch. Ek gaan nou so werk (P8).*

Een of ander vorm van beplanning tydens probleemoplossing was egter vir die meeste onderwysers belangrik aangesien leerders nie 'n probeer-en-tref-manier van probleemoplossing in *Scratch* moet toepas nie. Die aard van *Scratch* kan maklik tot gevolg hê dat probeer-en-tref-maniere aangeleer word (sien 2.3.1.2).

> *Want anders trek hulle alles [in met blokke kode] en sien dit werk nie en haal weer uit en sit weer in en dan is daar geen beplanning nie (P1).*

Onderwysers was egter onseker oor hoe om probleemoplossing en algoritme-ontwerp by *Scratch*-onderrig te integreer. *Scratch* is op die oog af 'n eenvoudige programmeertaal en dit is teoreties moontlik om oplossings te ontwikkel sonder om algoritme-ontwerp aanvanklik toe te pas. Dit is duidelik dat onderwysers nie die omvang en waarde van *Scratch* aanvanklik besef het nie.

> *Aanvanklik toe ek Scratch gesien het, het ek gedink dis 'n groot speletjie en eintlik 'n grap, want dis net insleep-programmering, maar toe ons nou 'n bietjie meer in die program ingaan, toe ek ook probeer het om 'n program te skryf, toe sien ek daar is darem lyf daaraan, dat hulle basiese konsepte gaan leer (P3).*

Uit bostaande kan die afleiding gemaak word dat onderwysers oor die algemeen glo dat algoritme-ontwerp noodsaaklik is. Sommige onderwysers het wel algoritme-ontwerp as 'n aparte tema hanteer en dit nie met die onderrig van *Scratch*-programmering geïntegreer nie. Dit is omdat die handboek dit net in een hoofstuk hanteer het en verskillende opinies bestaan het oor die plek van algoritme-ontwerp tydens die onderrig van programmering.

> *Toe ons handboeke gekry het, toe probeer ons weer teruggaan [na algoritmes]. Die kinders was toe al vir twee weke gewoond om in Scratch te werk en jy kan nie dit in daardie volgorde doen nie, glad nie (P6).*

Met al die bostaande in ag geneem, kan samevattend gesê word dat onderwysers algoritme-ontwerp en probleemoplossing as 'n integrale deel van die proses van programmering sien. Baie tyd word aan die onderrig van algoritmes en probleemoplossing afgestaan alhoewel almal dit nie geïntegreerd met *Scratch*-programmering onderrig nie. Sommige deelnemers het gevoel dat *Scratch* 'n maklike programmeertaal is en dat dit onnodig is om met algoritme-ontwerp te begin. Ander deelnemers het egter, ongeag die feit dat dit vir leerders moeilik en vervelig was, met algoritme-ontwerp begin en so gepoog om deursettingsvermoë en beplanning by leerders te kweek. Die feit dat onderwysers soveel verskillende strategieë probeer het, en van strategieë verwissel het, dui op die komplekse aard van die saak en bevestig die opvatting dat algoritme-ontwerp as die moeilikste fase van die programmeringsproses beskou kan word (sien 2.3). Uiteindelik het almal, met die uitsondering van deelnemer P7, tot die gevolgtrekking gekom dat algoritme-ontwerp, ook tydens die onderrig van *Scratch*, deurlopend toegepas moet word.

Die onderrig van probleemoplossing en algoritme-ontwerp is egter slegs een afdeling van die onderrig van programmering. Daar is verskeie ander programmerings-

beginsels en -begrippe wat onderrig moet word (sien figuur 2.9), soos wat ook uit die data-analise oor die onderrig van *Scratch*-programmering na vore gekom het.

4.2.3 Die onderrig van ander programmeringsbeginsels en -begrippe

Verskeie temas rakende die onderrig van programmeringsbegrippe in *Scratch* het tydens die onderhoude na vore gekom, en is met kodes beskryf. Om hierdie kodes te konseptualiseer, is dit in die volgende kategorieë (sien figuur 3.3) saamgevoeg: herhalingstrukture, besluitnemingstrukture, datatipes en veranderlikes, OOP en tekslêers. Die volgorde waarin die kategorieë genoem word, stem ooreen met die aantal kere wat deelnemers na die onderskeie kodes wat daarin vervat is, verwys het (gerangskik van dikwels genoem na selde genoem). Dit het spontaan uit gesprekke met deelnemers na vore gekom en stem ooreen met programmeringsbeginsels en –begrippe wat deur Maloney *et al.* (2008) en Meerbaum-Salant *et al.* (2013) genoem is wat die onderrig van programmeringsbeginsels of -begrippe in *Scratch* betref (sien 2.8.1).

In die meeste gevalle het slegs enkele van die kodes uit gesprekke met deelnemers na vore gekom.

In die volgende onderafdelings word oor die onderrig van programmeringsbegrippe en -beginsels in *Scratch* verslag gedoen.

4.2.3.1 Herhalingstrukture

Die meeste deelnemers het melding gemaak van herhaling as ’n programmeringsbegrip wat deur middel van *Scratch* onderrig kan word. Die rede is waarskynlik omdat *Scratch* op animasie gerig is en herhaling ’n groot rol speel by die beweging van *Sprites* (sien 2.4.4.5). Verskeie kodes rakende herhaling het uit die data-analise na vore gekom, naamlik beweging van *Sprites*, metodes vir die onderrig daarvan, onderskeid tussen verskillende herhalingstrukture, selfontdekking, leerders se begrip van herhalingstrukture en swak programmeringsgewoontes. Hierdie kodes word vervolgens verder bespreek.

Twee metodes vir die onderrig van herhaling is genoem. Eerstens het onderwysers reeds in *Scratch* na herhalingstrukture wat in *Delphi* voorkom, verwys.

> *In Delphi kry jy ’n WHILE-lus wat jy moet herhaal. In Scratch kry jy dieselfde, maar hulle maak dit net vir jou makliker, want jy hoef nie alles in te tik nie; jy trek [net blokke kode] in en dit wat jy dan wil herhaal, sit jy dan net in daardie stukkie in (P1).*

’n Volgende metode is dat leerders self die begrip *herhaling* moet ontdek, deur middel van ’n probleemscenario waarin herhaling gebruik moet word om die probleem op te los.

> *Ek sal vir hulle ’n probleem gee waar hulle byvoorbeeld ’n herhalende kode die heeltyd moet gebruik en dan sal ek vir hulle betreffende die oplossing sê, hier is ’n REPEAT, gebruik die REPEAT, dan hoef jy nie drie honderd en sestig keer iets te sê nie. Ek hou daarvan as hulle self die probleme ervaar (P8).*

In bostaande opmerking van deelnemer P8 was die bedoeling dat leerders self die herhaling moet ontdek, maar die onderwyser het egter reeds die oplossing aan hulle genoem en ook besluit watter soort struktuur gebruik moet word.

Alhoewel herhaling deur alle deelnemers geïdentifiseer is as ’n programmerings-begrip wat deur *Scratch* onderrig kan word, is daar daarop staatgemaak dat leerders herhaling intuïtief sal aanleer wanneer hulle in *Scratch* programmeer. Dit was egter duidelik dat leerders nie die diepere werking van herhalingstrukture verstaan het nie.

> *Scratch verdoesel ook eintlik dit wat agter die skerms gebeur (P6).*

> *Dis ook partykeer bietjie moeilik vir die kind, want hy weet nie altyd hoekom en waarom die instruksies bymekaarkom nie (P7).*

Slegs deelnemer P4 het kortliks na ’n onderskeid tussen voorwaardelike en onvoorwaardelike herhaling verwys.

> *Ek het byvoorbeeld vir hulle gesê as mens weet dat jy ’n aftelbare hoeveelheid data, of getalle het, gaan jy, wat ons nie pertinent in Scratch gedoen het nie, ’n FOR-lus gebruik. En ek het gepraat van WHILE en die REPEAT en FOREVER. Maar ek het vir hulle spesifiek gesê in Delphi gaan dit anders te werk. Jy gaan moet bepaal watter lus jy gaan gebruik (P4).*

Deelnemer P7 het soos P4 op die begrip van herhalingstrukture uitgebrei, naamlik dat die verskil tussen die onderskeie strukture aan leerders verduidelik moet word.

> *Ek het vooraf eers vir hulle gesê van die verskillende REPEATs wat jy kry, elkeen deurgegaan en gesê waarvoor jy elkeen gebruik en dan begin hulle en verstaan dit ook beter (P7).*

Aangesien leerders enige soort herhalingstruktuur in *Scratch* kan gebruik om ’n probleem op te los, ontstaan slegte programmeringsgewoontes (sien 2.4.2.2). Die

rede hiervoor is moontlik dat leerders die basiese begrip herhaling verstaan, en hulle programme op die oog af reg werk, sonder dat die beste herhalingstrukture gekies is. Daar bestaan verskeie herhalingstrukture in *Scratch* (sien 2.3.1.5.2), maar afgesien van deelnemer P7 het onderwysers nie aangedui dat hulle die verskille tussen die onderskeie herhalingstrukture aan leerders uitwys nie. Die interne werking van die strukture is dus nie aan leerders verduidelik nie en strukture is nie met mekaar vergelyk nie sodat die beste keuse vir die oplossing van ’n probleem nie gemaak kon word nie.

Alhoewel deelnemers dit dus eens was dat die programmeringsbegrip herhaling deur middel van *Scratch* onderrig kan word, is dit meestal aan selfontdekking oorgelaat. Leerders se aandag is in die meeste gevalle nie op die verskil tussen die onderskeie herhalingstrukture gevestig nie, alhoewel hulle die begrip herhaling intuïtief begryp het.

Die volgende kategorie waarvan die bevindings gerapporteer word, is die onderrig van keusestrukture en besluitneming.

4.2.3.2 Besluitnemingstrukture

Kodes rakende die onderrig van besluitnemingstrukture het nie so sterk soos die onderrig van herhalingstrukture na vore gekom nie en slegs enkele deelnemers het na die onderrig hiervan in *Scratch* verwys. Onderrig was in die meeste gevalle beperk tot eenvoudige besluitneming tussen twee keuses. Onderwyser P5 gee in die volgende aanhaling ’n aanduiding van hoe besluitneminstrukture onderrig is.

> *Wanneer ons die vraag analiseer, wys dit dat ons ’n bepaalde voorwaarde het wat ons moet toets. As hierdie voorwaarde waar is, dan gaan dit gebeur en as dit nie waar is nie, dan gaan iets anders gebeur (P5).*

Besluitnemingstrukture in *Scratch* is soortgelyk aan dié wat in *Delphi* voorkom, alhoewel daar nie ’n ekwivalent vir die CASE-stelling soos dit in *Delphi* gebruik word, in *Scratch* bestaan nie (sien 2.4.4.5). Onderwysers het nie melding gemaak van geneste keusestrukture of die onderrig van Boole-uitdrukkings nie, alhoewel dit deur die KABV vereis word (DBE, 2011:23). ’n Moontlike verklaring is weer eens dat dit nie pertinent onderrig is, of belangrik genoeg geag is om onderrig te word nie, of

aanvaar is dat leerders dit intuïtief sal aanleer wanneer hulle met *Scratch* programmeer.

Vervolgens word bevindings rakende kodes met betrekking tot die kategorie veranderlikes en datatipes bespreek.

4.2.3.3 Veranderlikes en datatipes

In *Scratch* word veranderlikes konkreet en visueel voorgestel en daarom verstaan leerders die begrip makliker in *Scratch* as in ander programmeertale (sien 2.4.4.4). *Scratch* verskaf dus die ideale omgewing om hierdie abstrakte begrip, wat oor die algemeen moeilik begryp word, maar vir beginnerprogrammeerders noodsaaklik is (Maloney *et al*., 2010:6), bekend te stel. Ten spyte hiervan het slegs twee deelnemers (P4 en P5) aspekte met betrekking tot datatipes en veranderlikes wat met *Scratch* onderrig kan word, geïdentifiseer. Kodes wat in hierdie kategorie na vore gekom het, was leerders se begrip van veranderlikes, die onderrig van veranderlikes in *Scratch*, die onderrig van datatipes in *Scratch* en leerders se begrip van datatipes in *Scratch*.

Onderwysers erken dat veranderlikes 'n belangrike programmeringsbeginsel is, maar het nie genoegsaam klem gelê op die onderrig daarvan ten einde begripvorming vas te lê nie. Sommige onderwysers het die onderrig van veranderlikes tydens algoritme-ontwerp aangespreek. Dit wou dus voorkom of onderwysers die programmeringsbeginsel van veranderlikes slegs bolangs verduidelik het en op leerders se intuïtiewe begrip van *Scratch* se konkrete en visuele voorstelling van veranderlikes staatgemaak het.

> *In Scratch is veranderlikes die beginsel wat baie, baie belangrik is (P5).*
>
> *Dit is vir hulle maklik om 'n veranderlike te skep, want hulle hoef nie 'n datatipe daarby te sit nie. Hulle weet hulle moet 'n veranderlike verklaar wat verskillende waardes kan aanneem. En hulle weet dit is 'n plek in die geheue wat geskep word, wat kan verander word met verskillende waardes. Hulle besef nou meer wat 'n veranderlike is (P4).*
>
> *As ek teruggaan na die TVA-tabel toe, dan sal ek sê dat ons veranderlikes nodig het om punt 1, punt 2 en punt 3 in te voer. Op hierdie stadium weet ons nie wat die waarde van punt 1 is nie, maar as jy 'n veranderlike kies wat jy kan stoor, maar ons weet nie wat die waarde is nie, dan sê ons net punt 1, punt 2 of punt 3 (P5).*

Die begrip van datatipes gaan hand aan hand met die beginsel van veranderlikes, soos tereg bo deur deelnemer P5 gemeld. Al is die begrip van datatipes deursigtig in *Scratch* (sien 2.4.4.4), bly dit 'n integrale deel van programmering (sien 2.3) en 'n belangrike begrip om te verstaan. Dit wil egter voorkom asof die onderwysers, met uitsondering van P4, nie datatipes aan leerders verduidelik het nie. Deelnemer P4 het bevestig dat sy wel na datatipes in *Scratch*-onderrig verwys en onderskeid tref tussen stringe en getalle, heelgetalle en reële getalle, en tussen karakters en stringe.

> *Ek sê vir hulle, onthou, 'n woord is altyd ['n string tipe]. Jy verklaar dit as 'n string en dan kry mens 'n karakter wanneer 'n string net beperk is tot een [karakter]. En dan het ek vir hulle vertel van heelgetalle en van reële getalle en as ons dit nou in 'n program gebruik, dan sal ek vir hulle sê alhoewel Scratch dit nou dadelik as 'n getal gesien het [moet hulle] onthou dat Delphi eintlik alles as 'n string sien en ons dan verskillende datatipes moet verklaar (P4).*

Hierdie deelnemer was van mening dat dit wel moontlik is om die begrip datatipes in *Scratch* te onderrig. Die onderwyser het weer eens 'n vooruitverwysing na *Delphi* gemaak. Nadat leerders se aandag op datatipes gevestig is, het hulle begin om begrip daarvan te toon.

> *Ja, hulle het definitief die begrip gehad dat 'n woord 'n woord is en 'n getal 'n getal is en dat 'n mens verskillende getalle kan kry (P4).*

Die gevolgtrekking kan dus gemaak word dat onderwysers hoofsaaklik op *Scratch* gefokus het vir die ontwikkeling van logiese denke en probleemoplossing en nie werklik daarvan bewus was dat *Scratch* 'n ware programmeertaal is (sien 2.4.4) waarmee programmeringsbegrippe, soos veranderlikes en datatipes, onderrig kan word nie. Geen deelnemers het in hul onderrig pertinent na aspekte soos reikwydte en benoemingskonvensies van veranderlikes, verwys nie. Tog word daar duidelik in die KABV (DBE, 2011:21) beskryf dat leerders vroeg in *Scratch*-onderrig reeds globale veranderlikes en lokale veranderlikes, benoemingskonvensies van veranderlikes en datatipes soos heelgetal-, string-, reële- en Boole-tipes moet verken.

Onderwysers het ook melding gemaak van objekgeoriënteerde begrippe met betrekking tot *Scratch*-onderrig, wat vervolgens bespreek word.

4.2.3.4 Objekgeoriënteerde programmering

Hierdie kategorie behels slegs die onderrig van objekgeoriënteerde begrippe met *Scratch*.

Alhoewel *Scratch* nie as ’n volwaardige objekgeoriënteerde programmeertaal gesien kan word nie, bevat dit verskeie analogieë na objekgeoriënteerde begrippe (sien 2.4.4.1). Ook in die KABV (DBE, 2011:21) word dit duidelik gestel dat die gebruik van animasiekarakters (*Sprites*) as objekte verken moet word. Daar kon egter min bevestiging van hierdie werkswyse by deelnemers gekry word. Slegs deelnemer P3 het vlugtig genoem dat basiese begrippe van objekte wel met *Scratch* onderrig kan word.

> *Basiese begrippe oor objekte kan jy vir hulle leer, soos die eienskappe van Sprites (P3).*

Die afleiding is dus gemaak dat onderwysers *Scratch* nie as ’n programmerings-omgewing om objekgeoriënteerde programmeringsbegrippe aan leerders te onderrig sien nie, wat teenstrydig is met die aanbevelings in die KABV. Daarvolgens moet objekgeoriënteerde begrippe reeds in graad 10 aan leerders bekend gestel word, alhoewel formele objekgeoriënteerde programmering eers in graad 12 in *Delphi* onderrig word. Die veronderstelling is dat daar reeds vanaf graad 10 stelselmatig op objekgeoriënteerde begrippe voortgebou moet word om leerders vir graad 12 voor te berei (DBE, 2011:12, 21). Aangesien die werkswyse nie gevolg is nie, bestaan die gevaar dat leerders in graad 12 in Noordwesprovinsie sal sukkel om die oorgang na ’n objekgeoriënteerde benadering te maak (sien 2.4.1.2).

Laastens word gerapporteer oor deelnemers se ervarings ten opsigte van die onderrig van tekslêers en lyste in *Scratch*.

4.2.3.5 Tekslêers en lyste

Alhoewel tekslêers nie in die literatuur geïdentifiseer is as ’n begrip wat deur middel van *Scratch* onderrig kan word nie, is dit waarskynlik omdat lyste of skikkings aanvanklik nie deel van *Scratch* was nie. Dit is eers later in *Scratch* 1.3 bygevoeg (MIT Media Lab, 2014a). Volgens die KABV moet leerders in staat wees om data in lyste as tekslêers in *Scratch* te stoor (DBE, 2011:26) vir invoer na ’n volgende

programuitvoering. Dit wil egter voorkom of slegs een onderwyser (P3) tekslêers by leerders probeer tuisbring het.

> *Dan het hulle stringhantering gedoen, waar hulle eintlik tekslêers moes gebruik het. Dan het ek vir hulle gesê jy gebruik eintlik Delphi. Dan het hulle tekslêers verstaan en hoe om 'n tekslêer te kan gebruik (P3).*

Bostaande aanhaling bevat weer 'n vooruitverwysing na *Delphi* en dit is onwaarskynlik dat die betrokke onderwyser se verduideliking daartoe sou lei dat leerders tekslêers kon gebruik.

In die volgende onderafdeling word tot 'n algemene gevolgtrekking van onderwysers se ervaring van die onderrig van programmeringsbeginsels in *Scratch* gekom.

4.2.3.6 Algemene gevolgtrekking oor die onderrig van programmerings-beginsels en -begrippe

Alhoewel baie aandag gegee is aan die onderrig van probleemoplossing en die ontwerp van algoritmes (sien 4.2.2) is programmeringsbeginsels en -begrippe nie pertinent onderrig nie en onderwysers het meestal staatgemaak op leerders se intuïtiewe begrip van programmeringsbegrippe in *Scratch*. Feitlik geen verwysing is na fouthantering en toetsing, wat 'n belangrike programmeringsbeginsel is (sien 2.4.5.3), gemaak nie. Onderwysers was onseker hoe om *Scratch*-onderrig te benader en het in *Scratch*-onderrig na *Delphi* verwys, aangesien laasgenoemde 'n bekende verwysingsraamwerk vir hulle was. Leerders het egter nog geen *Delphi*-kennis gehad nie en kon nie op hierdie verwysingsraamwerk voortbou nie.

> *Ek het baie keer teruggegaan na Delphi toe alhoewel hulle nog nie Delphi gedoen het nie (P1).*

Onderwysers het oënskynlik nie die KABV bestudeer om te bepaal wat met *Scratch* onderrig moet word nie, maar het blykbaar op handboeke en hulle eie interpretasie van *Scratch*-onderrig staatgemaak. Die gevolg was dat onderrig op *Scratch* gefokus het en nie op die aanleer van programmeringsbeginsels en -begrippe met *Scratch* nie. Begripvaslegging van veranderlikes, datatipes, herhaling- en besluitnemingstrukture, objekte, tekslêers en lyste was nie vir onderwysers van belang nie.

Bevindings rakende kodes wat met programmeringsbeginsels en -begrippe verband hou, is in die voorafgaande bespreking gerapporteer. Vervolgens word die kategorie onderrig–leerstrategieë en metodes wat met die onderrig van *Scratch* gevolg is, bespreek.

4.2.4 Onderrig–leerstrategieë en metodes met die onderrig van *Scratch*

Onderwysers het verskeie strategieë tydens die onderrig van *Scratch* toegepas. Die kodes wat in hierdie kategorie voorkom en wat vervolgens bespreek word, is direkte onderrig–leerstrategie, eksperimentering, selfontdekking en samewerking tussen leerders.

Om die navorsingsvraag te beantwoord, naamlik hoe *Scratch* onderrig word, wou die navorser nie net op die inhoudelike wat onderrig word, fokus nie, maar is daar ook na die onderrig–leerstrategieë wat tydens die onderrig van *Scratch* gevolg word, gekyk. Die aard van *Scratch* is sodanig dat leerders aangemoedig word om op ŉ eksploratiewe en eksperimentele manier te leer programmeer (sien 2.4.2). Die genot van *Scratch* lê verder in die sosiale aard daarvan, en *Scratchers* deel graag hulle programme met mekaar (sien 2.4.2). In die volgende aantal paragrawe word die bevindings rakende die onderrig–leerstrategieë wat onderwysers tydens *Scratch*-onderrig toegepas het, bespreek.

Deelnemers het nie onderrig–leerstrategieë (sien 2.7.1) by die naam genoem nie, maar het beskrywings van hulle strategieë gegee. Oor die algemeen het onderwysers ŉ tradisionele, direkte onderrig–leerstrategie toegepas. Hiermee word bedoel dat die deelnemende onderwysers gerapporteer het dat hulle die inhoud verduidelik, voorbeelde demonstreer, leerders voorbeelde afskryf en dan soortgelyke oefeninge doen. Ten spyte van die toepassing van ŉ direkte onderrig–leerstrategie het onderwyser P4 genoem dat leerders *Scratch* geniet en dat hulle graag hulle werk met mekaar deel. Die belangrikste, volgens P4, is dat leerders ondersteun moet word om in hulleself te glo, skeppend te wees en te verken. Alhoewel P4 mooi ideale gehad het met betrekking tot *Scratch*-onderrig, het sy steeds oorwegend ŉ direkte onderrig–leerstrategie toegepas.

Deelnemers het soms ook ander metodes soos eksperimentering en samewerkende leer, saam met die direkte onderrig–leerstrategie geïnkorporeer en leerders aangemoedig om op hul eie meer oor *Scratch* te lees en te ontdek.

> *Deur demonstrasie. Ek wys vir hulle 'n program en ek wys vir hulle verskillende programmeringsvoorbeelde, sodat hulle dit kan afskryf (P3).*
>
> *… dan sal ek net gou wys hoe werk 'n IF-stelling en daarna het hulle vrye teuels, dan kan hulle aangaan met wat hulle moet doen. Ek wys maar net vir hulle en dan moet hulle self probeer. As hulle vasbrand dan kan een van die klas, of ek, vir hulle wys maar ek sê nie spesifiek hoe dit gedoen moet word nie (P2).*
>
> *Ek sal dit op die dataprojektors en my skerm opsit en dan sal ek vir hulle verduidelik hoe [werk] die probleem wat ek vir hulle wil gee en dan sal ek die logika daarvan verduidelik. Dan sal ek die komponente begin opsit deur vir hulle te sê om hierdie by die komponente in te trek en te kyk wat gebeur, dan moet hulle vir my sê wat gaan gebeur en dan sal sê ek hoe ons dit op die spesifieke probleem gaan toepas. Dan kan hulle ook insae lewer voordat ek vir hulle sê hoe dit gedoen moet word. So, hulle moet ook bietjie gaan eksperimenteer (P1).*
>
> *Totáál informeel, ek betrek al die kinders. Hulle neem partykeer my hele les oor soos wat hulle mekaar verduidelik. Hulle het nie net slaafs gevolg wat in die klas gedoen was nie; hulle het bietjie gaan nalees en hulle kon meer instruksies byvoeg (P2).*
>
> *Ek sal byvoorbeeld vir hulle sê trek hierdie en hierdie komponent in, sit die Sprite op, trek die komponent op, wat gebeur met die Sprite? Dan moet hulle self besluit wat gebeur. Dan sal ek 'n nuwe probleem gee en vra wat hulle sal gebruik dat dit op dieselfde manier werk (P1).*
>
> *Ek gebruik voorbeelde uit die handboek. Ek sal vir hulle 'n voorbeeld demonstreer op die dataprojektor, ek sal die dataprojektor afsit en dan sal hulle dit as klas doen. Ek laat hulle nooit net begin nie; ek wys altyd eers. My kinders sê byvoorbeeld vir my ek hoef eintlik nie eers vir hulle te verduidelik nie; ek kan maar net dat hulle in hulleself glo. Hulle spog by mekaar oor wat hulle kan regkry en die een wil by die ander een sien hoe het hy dit reggekry. Die kinders geniet dit ongelooflik. Vir my is dit basies om skeppend te wees, om hulle eie limiete te oorskry. Om te verken, om verder te gaan verken – dit is wat ek nogal in Scratch sien (P4).*

Deelnemer P7 het die belangrikheid van selfontdekking genoem, alhoewel 'n direkte onderrig–leerstrategie ook weer toegepas is. Selfontdekking word skynbaar deur hierdie onderwysers gebruik vir die aanleer van eenvoudige begrippe, maar die moeiliker begrippe word eers aan leerders verduidelik.

> *Selfontdekking, ja selfontdekking. Ek gee vir hulle 'n probleem, dan sê ek vir hulle hulle moet die blokke self oortrek en kyk wat elkeen doen, tot hulle die regte patroon gekry het. As hulle nie weet nie, dan moet hulle die hulp-funksie gebruik. Die makliker aspekte, soos die beweging van die Sprite, laat ek hulle self ontdek, maar teoretiese aspekte sal ek deurtrek na Delphi toe (P7).*

> *Ek verduidelik eers die probleem vir hulle en sê dis wat ek wil hê en dan begin hulle programmeer. Soos wat elkeen werk, help ek elkeen om die regte rigting te kry en as ek sien hy is heeltemal verlore, dan gaan help ek van die begin af en sê kyk daarna, doen dit, of maak so (P7).*

Alhoewel koöperatiewe leer (sien 2.4.1.1) as onderrig–leerstrategie nie formeel toegepas is nie, was daar wel verwysings na samewerking tussen leerders.

> *Hulle mag met mekaar praat. Hulle mag mekaar help (P4).*

> *Dis maar die enigste manier – laat hom in 'n paar programmeer en saam met die leerder werk wat weet wat aangaan (P3).*

Deelnemer P3 het verwys na paarprogrammering en genoem dat sy dit probeer toepas. Dit was egter net 'n verwysing na samewerking tussen twee leerders, een wat die werk verstaan en een wat dit nie verstaan nie, en nie 'n behoorlike toepassing van al die nodige beginsels van hierdie onderrig–leerstrategie nie (sien 2.7.1).

Deelnemer P6 was die enigste onderwyser wat sterk gevoel het oor die toepassing van die direkte onderrigmetode en was van mening dat selfontdekking klasdissipline benadeel.

> *[In 2012] wou hulle net aangaan, wou net hulle eie goeters [hul eie Scratch-programmering] doen...Ek leer om hierdie jaar [2013] meer konserwatief skool te hou as net vry liberaal. Omdat ons handboeke het, is ek baie konserwatief in daardie opsig want om leerders net te los en die wiel self te ontwerp, is te moeilik in programmering. As jy nie in die begin van die jaar dissipline in die klas gevestig het nie, gaan jy dit verloor as jy later moeilik begrippe [afdelings van die werk] moet behandel. Jy gaan nie in graad 11 hierdie gedissiplineerde groep kry nie; hulle gaan nog steeds hulle truuks van graad 10-Scratch uithaal. Die kinders is [hierdie jaar] kalmer en hulle het wonder bo wonder pragtig algoritmes geskryf vir beide herhalingsprogramme en besluitneming. Hulle geniet dit, moenie 'n fout maak nie, hulle geniet die visuele (P6).*

Deelnemer P8 het nie 'n direkte onderrig–leerstrategie gebruik nie en het net op selfontdekking en eksperimentering gefokus.

> *Ek het vir hulle gesê om Scratch-tutoriale af te laai en solank by die huis daarmee te speel terwyl ek besig was met die teorie. Teen die tyd wat ek met die prakties [Scratch] begin het, het hulle al 'n redelike idee gehad waaroor dit gaan en dan het ek net vinnig deur die program se uitleg gegaan, en waar jy moet klik om wat te doen. Die meeste van die tyd verkies ek eksperimentering (P8).*

P8 het later in die onderhoud aangedui dat leerders die maklike begrippe verstaan het, maar gesukkel het om meer komplekse begrippe te verstaan.

> *Die eenvoudige begrippe verstaan hulle en wanneer jy begin kom by besluitneming en herhaling [dan sukkel hulle] (P8).*

Bogenoemde uitspraak van P8 stem ooreen met die uitsprake in die studie deur Meerbaum-Salant *et al*. (2013:244) (sien 2.8.1) waar bevind is dat leerders wel bepaalde programmeringsbeginsels en -begrippe self met *Scratch* kan ontdek, maar dat verdere riglyne vir die aanleer van ander beginsels en begrippe benodig word.

Die rapportering van strategieë wat tydens die onderrig van *Scratch* toegepas is, kan soos volg saamgevat word. Alhoewel ’n direkte, tradisionele onderrig–leerstrategie oor die algemeen deur onderwysers toegepas is, is daar ook op indirekte wyse melding gemaak van die behoefte aan ander onderrig–leerstrategieë soos koöperatiewe leer en eksperimentering. Die aard van *Scratch* bevorder op ’n natuurlike manier sosiale interaksie en selfgerigtheid onder leerders. Leerders wil saamwerk, hulle probleme en oplossings met mekaar deel en meer selfgerig werk, maar sommige onderwysers het aangedui dat dit ’n bedreiging vir klasdissipline inhou.

’n Verdere saak wat skynbaar vir onderwysers ’n kwessie was tydens die onderrig van *Scratch*, was die hoeveelheid tyd wat hulle aan *Scratch*-onderrig moes bestee en die vraag of hulle reeds in graad 10 met *Delphi*-onderrig moes begin. Dit word vervolgens in 4.2.5 bespreek.

4.2.5 Tyd wat aan *Scratch*-onderrig bestee word

Deelnemende onderwysers was van mening dat hulle in graad 11 en graad 12 dieselfde hoeveelheid werk moes doen wat hulle voorheen in drie jaar (graad 10 tot 12) gedoen het. Volgens hulle het hulle te min tyd gehad vir *Delphi*-onderrig.

Die meeste onderwysers het gerapporteer dat hulle die volle 2012 aan *Scratch* bestee het. Die redes wat genoem is, was dat hulle laat begin het met *Scratch* as gevolg van die handboekkwessie, dat die volume werk sodanig was dat dit hulle die hele jaar besig gehou het, en dat daar probleme met rekenaartoerusting was.

> *Ek was een van die negatiewes wat gedink het ek gaan ’n kwartaal of twee Scratch doen, klaarmaak en met Delphi aangaan, maar ek kon nie; daar was net te veel werk. Ek het ongeveer tot die derde, vierde kwartaal nog Scratch gedoen (P1).*

> *Ek het glad nie begin met Delphi verlede jaar nie, nee. Ek het die laaste kwartaal net vir die PAT aangewend, so ek het glad nie, glad nie [met Delphi begin nie] en ek het baie gesukkel met rekenaartoerusting (P4).*
>
> *My hoof het gesê ek moet in graad 10 met Delphi begin, maar omdat ek so lank gewag het vir die handboeke, het ek redelik vasgevang geraak met konsepte wat ek hulle nog wou leer. Hy [die hoof] het gesê ons moet met Delphi begin, want Scratch is te maklik. Soos ek met die kinders gevorder het, het ek agtergekom dat Scratch my kinders daardie probleemoplossingsvaardighede gaan leer wat hulle nie het nie (P3).*

Tydens die eerste onderhoude is dus vasgestel dat programmeringsbegrippe nie pertinent onderrig is nie. Verskillende strategieë is gevolg tydens die onderrig van *Scratch* en onderwysers was onseker oor watter strategie gevolg moet word. Die gevolg was dat verskillende onderrig–leerstrategieë probeer is, waarvan die algemeenste ŉ direkte onderrig–leerstrategie was. Al die deelnemers het aangedui dat hulle die volle graad 10 jaar vir *Scratch* gebruik het en nie voor graad 11 met *Delphi* begin het nie.

Vervolgens word die resultate van die tweede onderhoude en die ondersoek van navorsingsvraag vyf – hoe onderwysers die oorgang na *Delphi* ervaar het – gerapporteer.

4.3 ONDERWYSERS SE ERVARING VAN DIE OORGANG NA *DELPHI*-ONDERRIG

Die doel met navorsingsvraag ses was om te bepaal hoe onderwysers die oorgang vanaf *Scratch*-onderrig na *Delphi*-onderrig ervaar het. Dié vraag is deur middel van ŉ tweede stel onderhoude met dieselfde deelnemers (sien 3.5.2) ondersoek. Die kategorieë wat uit die kodes ontstaan het en wat bespreek word, is –

- algemene aspekte met betrekking tot die oorgang van *Scratch* na *Delphi*; en
- die oordrag van programmeringsbeginsels en-begrippe vanaf *Scratch* na *Delphi*.

4.3.1 Algemene aspekte met betrekking tot die oorgang van *Scratch* na *Delphi*

Kodes wat in hierdie kategorie voorgekom het, was die volume werk van *Delphi* in graad 11, die *Delphi*-programmeringsomgewing en die kwessie van te min tyd.

Alhoewel *Scratch* en *Delphi* beide programmeertale is, is daar beduidende verskille tussen die aard van *Scratch* (sien 2.4.2) en die aard van *Delphi* (sien 2.5.2). Die aanpassing na *Delphi* was gevolglik groot, dit het meer tyd geneem as wat verwag is, ŉ groot volume werk moes gedek word, en leerders het gesukkel om die sprong van *Scratch* na *Delphi* te maak.

> *Ja, dit [Delphi] was ŉ skok (P6).*
>
> *Dit gaan nie goed met my graad 11's nie en ons is ongelooflik agter. Ek gaan nou volgende kwartaal Maandaemiddae ook skoolhou, maar my graad 11's sukkel regtig baie (P4).*

In die volgende onderafdeling word onderwysers se ervaring ten opsigte van die volume werk wat afgehandel moes word, bespreek.

4.3.1.1 Die volume werk van Delphi in graad 11

Die volume werk wat volgens die KABV (DBE, 2011:13) in die eerste ses maande afgehandel moes word, was vir die deelnemende onderwysers net te veel en hulle was gespanne daaroor. Die onderwysers was bekommerd omdat hulle slegs twee jaar gehad het vir *Delphi*-onderrig, alhoewel dieselfde inhoud voorheen in graad 10, 11 en 12 onderrig is (sien tabel 1.1).

> *Wat ons voorheen drie jaar gevat het om met die kinders reg te kry, druk hulle [KABV] in twee, of een-en-ŉ-halwe jaar in (P8).*
>
> *As ek so opgestres is teen die tempo wat ek moet werk, hoe moet hulle nie voel wat dit as ŉ nuwe vak neem nie. Ek voel jammer vir hulle, veral vir die leerders wat wil goed doen. Dis vir hulle baie swaar (P1).*

Alhoewel die *Delphi*-omgewing nuut was, moes bepaalde programmeringsbeginsels en -begrippe in *Scratch* reeds in graad 10 vasgelê word (sien tabel 1.1). Onderwysers het egter met graad 11 *Delphi*-onderrig gevoel dat hulle weer van nuuts af moes begin met die onderrig van programmering. Dit bevestig die bevindings van hierdie studie (sien 4.2) dat onderwysers nie *Scratch* aan leerders onderrig het met die doel om programmeringsbeginsels en –begrippe vas te lê nie. Hulle het nie besef dat *Scratch* wel meer as net probleemoplossing en ŉ liefde vir programmering kan bevorder nie.

> *Die kinders het dit [Delphi] heeltemal gesien as ŉ nuwe program. Integer, Boolean en daardie begrippe moes ek van vooraf met hulle doen (P7).*

Terwyl onderwysers gevoel het hulle begin van voor af met die onderrig van programmering in graad 11, word in die KABV (DBE, 2011:12) aangeneem dat leerders reeds in graad 10 bepaalde programmeringsbeginsels en -begrippe bemeester het. In graad 11 moes *Delphi*-onderrig slegs hierop voortbou (sien tabel 1.1). Daar is reeds in hierdie navorsing bevind dat programmeringsbeginsels en -begrippe nie behoorlik aan leerders in *Scratch* onderrig is nie (sien 4.2.3.6) en die deelnemende onderwysers het gevolglik paniekerig begin raak oor die tyd wat beskikbaar was om *Delphi* te onderrig.

> *Wat ons nou moes gedoen het, is basies om die hele [vorige] graad 10-sillabus in vier maande te doen (P4).*
>
> *Soos ons aangaan met die konsepte raak dit 'n panieksaaiery, want hulle sukkel met al die konsepte (P3).*

Behalwe die groot volume werk wat afgehandel moes word, moes leerders ook die nuwe *Delphi*-programmeringsomgewing aanleer. Die ervaring met betrekking tot die oorgang na die nuwe *Delphi*-proggrammeringsomgewing word vervolgens bespreek.

4.3.1.2 Die Delphi-programmeringsomgewing

Die deelnemers se ervarings rakende die *Delphi*-koppelvlak (sien 2.5.3), die sintaksis (sien 2.5.4.2) en fouthantering (sien 2.5.5.3) word vervolgens bespreek.

Die Delphi-koppelvlak

Die deelnemers het gerapporteer dat die *Delphi*-koppelvlak aanvanklik vir leerders vreemd was, maar hulle het met verloop van tyd daarby aangepas en geleer om daarmee te werk.

> *In die begin was dit baie vreemd; dit was nogal moeilik; dit is nie 'n maklike koppelvlak nie (P3).*
>
> *… deur die helfte van die tweede kwartaal het dit begin makliker raak vir hulle (P2).*

Aangesien *Scratch* egter 'n visuele programmeertaal is (sien 2.4.2) waar blokke kode slegs ingesleep word om programmeringskode mee te bou en *Delphi* 'n sintaksisgebaseerde taal (sien 2.5.2), was die verwagting dat die radikale verskille 'n probleem sou wees tydens die oorgang na *Delphi*. Die gevolgtrekking kon egter gemaak word dat die oorgang na die *Delphi*-koppelvlak met verloop van tyd nie vir

leerders ’n probleem was nie, maar dat dit wel tyd geneem het om leerders daarmee vertroud te maak.

Die sintaksis

Deelnemers P4, P5 en P6 was van mening dat leerders met die sintaksis sukkel. ’n Algemene klagte wat telkens voorgekom het, was dat leerders vergeet het om BEGIN- en END-instruksies in programmeringstrukture in te voeg.

> *Hulle vergeet om die BEGIN en END in te tik as daar meer as een stelling na die dubbelpunt is. Dit het ek gesien, maar dit is ’n klein foutjie (P4).*
>
> *Die enigste ding is die aanhalingstekens, daar is te veel aanhanlingstekens in Delphi in vergelyking met Scratch, maar leerders is nou gewoond daaraan. Sommige leerders, veral dié wat visueel afhanklik is, vind dit tot vandag toe nog moeilik om by te hou met Delphi waar alle instruksies getik moet word (P5).*
>
> *…hulle moes binne ’n kwartaal alles en sintaksis leer. Sintaksis bly vir nuwe gebruikers ’n uitdaging, ’n hekkie (P6).*

Die meeste deelnemers het egter nie gevoel dat die aanleer van sintaksis in *Delphi* vir leerders moeilik was nie. Algemene foute betreffende sintaksis, soos om aanhalingstekens en kommapunte te vergeet, is deur die meeste deelnemende onderwysers nie as struikelblokke gesien nie, maar as die normale verloop van die aanleer van ’n sintaksisgebaseerde programmeertaal. Deelnemers P1, P7, P3 en P8 het vertel dat leerders die sintaksis relatief maklik aangeleer het, al was dit aanvanklik vir hulle vreemd.

> *Ek dink hulle het dit [sintaksis] vir my vinnig opgevang, ja (P1).*
>
> *Hulle het dit [die sintaksis] nogal vinnig opgetel (P7).*
>
> *Dit was nogal 'n groot probleem aanvanklik gewees, maar nou is dit nie meer so nie (P3).*
>
> *Ek sou sê die sintaksis is vir hulle nie ’n probleem nie. Van hulle kan dit presies reg doen maar natuurlik is programmering maar 'n frustrerend as jy nie gewoond is aan die sintaksis nie (P8).*

Sintaksis word algemeen in die literatuur as ’n struikelblok by die aanleer van programmeertale vir beginners gesien (sien 1.1.1). Die bevindings van hierdie navorsing was egter dat die sintaksis nie vir graad 11-leerders die grootste struikelblok was nie. ’n Moontlike verklaring hiervoor lê in die bevindings van Jenkins (2002:55) en Resnick (2012) opgesluit. Hulle voer aan dat die skryf van

programmeringskode slegs ’n klein deeltjie van programmering behels en dat die prosesse van programmering (sien figuur 2.1) aangeleer moet word tydens die onderrig van programmering. Aangesien leerders reeds in *Scratch* met die prosesse van programmering kennis gemaak het, het hulle nie die sintaksis van *Delphi* as ’n probleem ervaar nie, maar slegs as ’n nuwe faset van programmering wat hulle op daardie stadium kon oorbrug. Die beginsel van fouthantering was egter nie vir hulle so maklik nie.

Fouthantering

Leerders is feitlik nooit in *Scratch* blootgestel aan programme wat nie wou uitvoer nie (sien 4.2.3.6) en daarom was dit, volgens die deelnemende onderwysers, vir leerders moeilik om foutopsporing te doen.

> *Hulle het nooit daarmee kennis gemaak nie. Nou as die program nie wil uitvoer nie, dan roep almal en almal het ’n probleem en niemand weet hoekom nie (P7).*
>
> *Hulle reageer nie, dis asof hulle so skrik as die program nie wil werk nie, dat alles vassteek (P4).*

Soos wat leerders meer blootstelling aan fouthantering gekry het, kon hulle wel die sintaksisfoute opspoor, maar hulle het steeds gesukkel om logiese foute op te spoor.

> *Hier is ’n paar van hulle wat presies agterkom as hulle daardie foute kry, dit aanteken, sodat hulle nie meer met die verwysing sukkel nie (P3).*
>
> *Hulle het nie daardie ervaring om regtig foute op te spoor nie, so foutopsporing is ’n probleem. (P6).*

Die gevolgtrekking word gemaak dat fouthantering ’n struikelblok was wat leerders in die *Delphi*-programmeringsomgewing ervaar het. Alhoewel sintaksisfoute nie in *Scratch* voorkom nie, kan logiese foute steeds algemeen in *Scratch*-programme voorkom (Maloney *et al*., 2010:5–6). Die aanname kan gevolglik gemaak word dat leerders reeds in *Scratch* blootstelling moes gekry het aan programme wat uitvoerfoute gee en toetsing van programme (DBE, 2011:23) om ’n grondslag te lê vir foutopsporing in *Delphi*. Foutopsporing is juis ook een van die prosesse van programmering wat onderrig moet word (sien figuur 2.9), ongeag die taal waarin programmering plaasvind.

> *Jy kan mos nie sê dis Delphi of Scratch nie, dis maar die onderwyser, hoe hy algoritmes, soek van foute en naspeurtabelle aan leerders oorgedra het (P6).*

Alhoewel die onderwysers baie aandag aan probleemoplossing en algoritme-ontwerp in *Scratch* gegee het (sien 4.2.2), het hulle nie pertinent aandag aan foutopsporing gegee nie. Dit kan as ’n struikelblok vir die oorgang na *Delphi* geïdentifiseer word.

Die aanleer van die *Delphi*-koppelvlak, sintaksis en fouthantering het waardevolle onderrigtyd in beslag geneem en veroorsaak dat onderwysers nie soos deur die KABV voorgeskryf is dadelik met gevorderde programmering in *Delphi* kon begin nie. Vervolgens word die kwessie van te min onderrigtyd bespreek.

4.3.1.3 Te min tyd vir Delphi-onderrig

Met bogenoemde bespreking van die oorgang na die *Delphi*-programmeringsomgewing, sowel as algemene aspekte rakende programmering in ag geneem (sien 2.3), is die gevolgtrekking gemaak dat die oorgang na die *Delphi*-programmeringsomgewing aanvanklik moeilik was, maar dat leerders wel later aan die omgewing gewoond geraak het. Dit het leerders egter langer geneem om die *Delphi*-programmeringsomgewing gewoond te raak as waarvoor in die KABV (DBE, 2011:31) voorsiening gemaak is. Daar word volgens die KABV verwag dat leerders dadelik moet begin voortbou op programmeringsbegrippe wat in graad 10 vasgelê moes gewees het (DBE, 2011:13, 31) en tyd is nie toegeken om aan die nuwe omgewing gewoond te raak nie.

> *Dit het hulle lank gevat om die Delphi-omgewing te leer en omdat dit so nuut was, maak nie saak as jy vir hulle sê, moenie te veel aandag hieraan gee nie. Dit vat tyd, dit vat regtig tyd, dit vat langer tyd as wat mens dink, dit het my ten minste drie weke gevat (P6).*

Onderwysers het gevolglik ’n behoefte gehad aan meer onderrigtyd en het in die middae en oor naweke skoolgehou.

> *Ek gaan nou volgende kwartaal Maandaemiddae ook skoolhou (P4).*
>
> *Die kwantiteit wat hulle in so ’n kort tydjie moet leer … ek kan nie dink hoe ek hulle vir matriek op datum sal kry nie! Ek sal tien teen een hierdie jaar nog voor die eindeksamen naweke moet inwerk (P8).*

Die behoefte aan meer onderrigtyd, die groot werkslading wat voorgeskryf is (DBE, 2011:30–40), tesame met die bevinding dat programmeringsbeginsels nie in graad 10 behoorlik onderrig is nie (sien 4.2.3.6), het baie spanning op onderwysers geplaas.

Met die graad 11's gaan dit goed, maar met my en die graad 11's gaan dit nie so goed nie (P2).

Vervolgens word bevindings met betrekking tot die oordra van programmerings-beginsels en -begrippe vanaf *Scratch* na *Delphi* bespreek.

4.3.2 Die oordrag van programmeringsbeginsels en -begrippe vanaf *Scratch* na *Delphi*

Wat die oordrag van programmeringsbeginsels en -begrippe betref, is daar positiewe terugvoer gelewer, naamlik dat oordrag wel plaasgevind het, maar die onderwysers het ook ervaar dat sommige programmeringsbeginsels en -begrippe nie na *Delphi* oorgedra is nie. Kodes wat in hierdie kategorie voorkom, is programmerings-beginsels en -begrippe wat wel oorgedra is, die leemtes wat opgemerk is, programmeringsbegrippe wat nie oorgedra is nie, en die redes wat daarvoor aangevoer is.

4.3.2.1 Oordra van programmeringsbeginsels en -begrippe

Dit was onderwysers se ervaring dat *Scratch* wel leerders gehelp het om *Delphi* vinniger te verstaan, aangesien logiese denke en probleemoplossing, sowel as sommige programmeringsbeginsels en –begrippe aan hulle bekend was.

Scratch het leerders gehelp om Delphi vinniger te verstaan (P1).

Vorige jare toe ek Delphi aan graad 10's onderrig het, moes ek alles onderrig. Noudat ek begin het met Delphi-onderrig, kom ek agter dat hulle reeds van sommige aspekte weet (P5).

Aangesien hierdie opmerking van deelnemer P5 dalk teenstrydig mag klink met vorige bevindings (sien 4.2.3.6), moet hier genoem word dat deelnemers P1 en P5 geïdentifiseer is as onderwysers wat positief is rakende die oordrag van begrippe vanaf *Scratch* na *Delphi* en wat gepoog het om programmeringsbeginsels reeds in *Scratch* te onderrig, soos wat later in die bespreking na vore sal kom.

Vervolgens word die spesifieke programmeringsbeginsels en -begrippe wat na *Delphi* oorgedra is, bespreek.

Logiese denke en probleemoplossing

Die meeste onderwysers was dit eens dat *Scratch* leerders se logiese denke en probleemoplossingsvermoë bevorder het en dat hulle hierdie vaardighede na *Delphi* kon oordra.

> *Hulle kan dit wat hulle van probleemoplossing geleer het, toepas (P5).*

Hierdie bevinding, naamlik dat die programmeringsbeginsel van logiese denke na *Delphi* oorgedra is, word daaraan toegeskryf dat die deelnemers in die algemeen baie klem gelê het op die onderrig van probleemoplossing en algoritme-ontwerp in *Scratch*-onderrig (sien 4.2.2). Deelnemer P8 was die enigste onderwyser wat nie in 2012 algoritme-ontwerp onderrig het nie, en was ook die enigste deelnemer wat die opmerking gemaak het dat algoritme-ontwerp vir graad 11-leerders problematies was.

Besluitneming- en herhalingstrukture

Die meeste oordrag is met betrekking tot besluitneming- en herhalingstrukture gerapporteer. Volgens onderwysers het leerders 'n basiese begrip van bogenoemde programmeringsbeginsels in *Delphi* getoon, maar 'n diepe begrip daarvan het ontbreek.

> *Die IF-stelling het hulle baie maklik verstaan (P3).*
>
> *Ek dink as 'n mens die FOR-lus of die REPEAT verduidelik het, was dit vir my of hulle dadelik gesnap het (P1).*
>
> *Die leerders het dit reggekry om hul kennis van Scratch effektief te herroep, om saamgestelde Boole-uitdrukkings te skep (P5).*

Wat besluitneming betref, het sommige onderwysers aangedui dat sodra die geneste-IF, die CASE-stelling en meer komplekse Boole-uitdrukkings behandel is, leerders daarmee gesukkel het.

> *Met die uitbreiding van die IF, waar ons AND en OR inbring en met die geneste IF het hulle bietjie gesukkel maar die konsep was dieselfde, hulle het dit gesnap. Ek weet hulle het baie met AND- en NOT-operators gesukkel (P3).*

Net so, met herhaling, het leerders gesukkel met meer ingewikkelde begrippe soos die werking van lusbeheerveranderlikes en die toepassing van verskillende soorte lusstrukture, alhoewel hulle die begrip van herhaling verstaan het,

Hulle verstaan herhaling, maar die lusbeheerveranderlike van die FOR-lus, is byvoorbeeld vir hulle 'n krisis (P6).

Al wat hulle regtig onthou is REPEAT. Hulle wil amper altyd na REPEAT toe gaan in Delphi (P3).

Veranderlikes en datatipes

Alhoewel leerders reeds in *Scratch* met veranderlikes kennis gemaak het, was dit duidelik dat hulle dit nie goed begryp het nie en gesukkel het om datatipes wat aan veranderlikes toegeken word, die omskakeling van datatipes en begrippe met betrekking tot reikwydte van veranderlikes te begryp. Leerders het ook die terminologie met betrekking tot die gebruik van veranderlikes en datatipes as vreemd ervaar.

Hulle het die begrip makliker begryp, maar hulle het nog steeds nie verstaan hoekom ons veranderlikes gebruik nie (P3).

Die konsepte van Integer, Boolean en String het 'n rukkie geneem om mooi te verduidelik (P7).

Veral die omskakeling tussen die verskillende datatipes, soos byvoorbeeld IntToStr, is nog steeds vir hulle moeilik; ek weet hulle sukkel daarmee (P3).

Daar is dele waar jy kan sê: "Hier is 'n tendens". Soos om veranderlikes op verkeerde plekke te verklaar (P6).

Integer is dalk vir 'n Afrikaanse kind 'n moeilike [woord], dis 'n nuwe woord (P4).

Uit die eerste onderhoude het dit duidelik geblyk dat onderwysers nie genoegsaam aandag geskenk het aan die vaslê van begrippe rondom veranderlikes en datatipes nie. Die gevolg daarvan was dat leerders hierdie begrippe nie kon oordra na *Delphi* nie en hulle gevolglik hierdie basiese beginsels van programmering eers in graad 11 moes aanleer.

Slegs P6 het aangedui dat die gebruik van veranderlikes in *Scratch* dit vir leerders in *Delphi* makliker gemaak het.

Die gebruik van veranderlikes in Scratch het dit definitief vir kinders makliker gemaak [in Delphi] (P6).

Samevattend kan gesê word dat, alhoewel die begrippe besluitneming en herhaling in die literatuur geïdentifiseer is (sien 2.4.4.5) as programmeringsbeginsels wat leerders in *Scratch* op hulle eie kan ontdek en gevolglik maklik kan verstaan, dieselfde nie vir veranderlikes en datatipes geld nie (sien 2.4.4.4 en 2.8.1). Meeste

van die deelnemers het ook die bevindings bevestig. ’n Diepere begrip van programmeringsbegrippe het by leerders ontbreek. Leerders het gesukkel met die geneste IF-stelling, die CASE-stelling, komplekse Boole-uitdrukkings waarin die operators AND, OR en NOT gebruik word, die onderskeid van soorte herhaling, lusbeheerveranderlikes, datatipes en reikwydte van veranderlikes.

Soos met die eerste onderhoude (sien 4.2.3.1 en 4.2.3.2), het die tweede onderhoude bevestig dat programmeringsbeginsels en -begrippe rakende keusestrukture en herhaling nie altyd op ’n behoorlike wyse aan graad 10-leerders onderrig is nie. Daar was wel uitsonderings waar deelnemers gerapporteer het dat programmeringsbeginsels oorgedra is, maar dit het plaasgevind omdat die onderwyser tydens *Scratch*-onderrig meer klem daarop geplaas het.

> *Hulle verstaan die begrip [besluitneming], omdat ek klem daarop gelê het in Scratch (P7).*

Objekgeoriënteerde programmeringsbeginsels

Deelnemer P3 het in die eerste onderhoud genoem dat basiese begrippe oor objekte reeds in *Scratch* aan leerders onderrig word. Tydens die tweede onderhoud was dit dieselfde deelnemer se ervaring dat graad 11-leerders wat in *Scratch* onderrig is, ’n beter begrip van objekte en die eienskappe daarvan gehad het, as leerders wat nie voorheen in *Scratch* onderrig is nie. Geen ander deelnemer het melding gemaak van objekte nie.

> *Normaalweg het die kinders altyd daarmee [met objekte] gesukkel en nou het ek vir hulle gewys en hulle kon dit daarna op hulle eie ook regkry. So daardie afdeling dink ek, is heeltemal ’n maklike oorskakeling vanaf Scratch na Delphi toe. Dis maar hulle geaardheid en instelling teenoor objekte, want hulle kan nou die eienskappe verander en hulle kan toepas wat hulle in Scratch ook kon doen (P3).*

Hier het dus ’n positiewe oordrag van die begrippe van objekte en eienskappe na *Delphi* plaasgevind, ten spyte daarvan dat dit vir die graad 10-leerders aanvanklik ’n baie vreemde begrip in *Scratch* was.

Tekslêers en skikkings

Leerders kon volgens onderwysers se mening die begrip tekslêers na *Delphi* oordra. Dit was egter steeds moeilik om die begrip in *Delphi* toe te pas.

> *Hulle het dadelik, maklik oorskakeling gedoen, veral met tekslêers, maar hoe ons dit gedoen het [in Delphi] was vir hulle ook maar weer 'n skok (P3).*

Deelnemer P1 se leerders was in staat om self assosiasies te maak en hulle begrip van lyste in *Scratch* na skikkings in *Delphi* oor te dra.

> *Maar wat ek wel gesien het toe ek skikkings begin verduidelik het, ek het nie eers gedink om vir hulle te sê onthou dis nou lyste in Scratch nie, en toe ek dit begin verduidelik, toe sê 'n kind: "O, dis soos lyste". So my kop is nog nie eers heeltemal daar om Scratch en Delphi bymekaar te bring nie. Ek begin net met skikkings en hier kom die kind en hy sê: "O, dis soos lyste" (P1).*

Leerders kon dus in hierdie geval die begrip lyste in *Scratch* na skikkings in *Delphi* oordra terwyl die onderwyser self nie daaraan gedink het om onderrig op hierdie oordrag te rig nie. Net so was daar ander programmeringsbeginsels en -begrippe wat nie in *Scratch* onderrig is met die oog op die oorgang na *Delphi* nie. Dit word vervolgens bespreek.

4.3.2.2 Programmeringsbeginsels- en begrippe wat nie oorgedra is nie en redes daarvoor

Onderwysers het verwys na enkele begrippe wat nie na *Delphi* oorgedra is nie, soos byvoorbeeld die bewerkingsoperator, MOD. Onderwysers het bepaalde redes daarvoor verskaf, naamlik dat hulle *Scratch*-onderrig nie spesifiek daarop gerig was nie en dat hulle nie hulle *Scratch*-onderrig beplan het met die oog op oordrag na *Delphi* nie.

> *Ek dink dit is vir hulle 'n heeltemal vreemde begrip. Ja, maar tog dink ek, ek het dit te min bespreek in Scratch. Hulle het MOD ['n bewerkingsoperator] in Scratch gehad, maar ek het nie soveel klem daarop gelê nie (P4).*

Nie al die onderwysers het in hulle onderrig beplan vir die oordrag vanaf *Scratch* na *Delphi* nie.

Deelnemers P3 en P5 was bekommerd oor die hoeveelheid werk wat in graad 11 afgehandel moet word, maar het leerders 'n voorsprong gegee in die sin dat hulle pertinent probeer het om aan te sluit by programmeringsbegrippe wat in graad 10 onderrig is en aan leerders die oordrag daarvan na *Delphi* aan te toon.

> *Ons trek by skikkings en tekslêers, dis waar ons moes wees vir hierdie eksamenvraestel, en dis baie werk om in 6 maande se tyd in te druk (P3).*

So, jy herinner hulle net vinnig daaraan, dit is nie dat jy dit weer vir hulle moet leer nie. Ek sien die waarde van Scratch. (P5).

Die gevolgtrekking word gemaak dat begrippe wel vanaf *Scratch* na *Delphi* oorgedra kan word en dat die oordrag verhoog kan word indien begrippe pertinent in *Scratch* onderrig word. Die onmiddellike voordeel van sodanige oordrag is ’n besparing in onderrigtyd – wat die deelnemers se groot dilemma ten opsigte van *Delphi*-onderrig was. Aangesien *Scratch* nie altyd pertinent onderrig is met die oog op die oordrag van programmeringsbegrippe na *Delphi* nie, was die implikasie dat onderwysers, wat reeds onder druk was met onderrigtyd (sien 4.3.1.3) onder nog groter druk verkeer het omdat programmeringsbeginsels weer van voor af onderrig moes word.

Behalwe dat onderrig nie volledig deur al die onderwysers op die oorgang na *Delphi* gerig was nie, het onderwysers ook verskeie ander redes aangevoer vir probleme wat hulle tydens *Delphi*-onderrig ervaar het, soos wat vervolgens bespreek word.

4.4 ANDER PROBLEME MET *DELPHI*-ONDERRIG

Hierdie kategorie bestaan uit kodes wat nie direk van toepassing is op die navorsingsvrae nie. Dit word nogtans gerapporteer aangesien dit ook uit die data wat ingesamel is na vore gekom het en steeds relevant is met betrekking tot die onderrig van *Scratch* en die oorgang na *Delphi*. Die kodes behels ’n nuwe leerervaring vir onderwysers, die ontoereikendheid van handboeke, leerders se akademiese vermoëns en ’n onderrig–leerstrategie vir *Delphi*-programmering.

Uit die bespreking tot dusver (sien 4.2.3.6 en 4.3.2) behoort dit duidelik te wees dat waar daar ’n gebrek aan pertinente onderrig van programmeringsbeginsels en -begrippe met die oog op die oordrag na *Delphi* was, negatiewe implikasies ten opsigte van deelnemende onderwysers se onderrig van *Delphi* waarneembaar was.

Van die redes wat hiervoor verskaf word, is nie direk van toepassing op die onderrig van *Scratch* nie. Dit word egter steeds genoem te einde onderwysers se persepsies van die oorsake van probleme uit te lig, aangesien die blaam in baie gevalle nie op die onderrig van *Scratch* geplaas word nie. Die redes, soos deur onderwysers aangevoer, vir leerders se gebrek aan begrip van programmeringsbegrippe word vervolgens gerapporteer.

4.4.1 ’n Nuwe leerervaring vir onderwysers

Daar is reeds in 4.2.1 gerapporteer dat onderwysers onseker was oor *Scratch*-onderrig. Hierdie onsekerheid is na *Delphi*-onderrig oorgedra en onderwysers het dit in die tweede onderhoude as rede aangevoer vir probleme met *Delphi*-onderrig. Omdat *Scratch* vir hulle ’n nuwe leerervaring was en aangesien hulle nie geweet het hoe om die onderrig te benader nie, het dit tot probleme in *Delphi*-onderrig gelei. Al was van hulle ervare IT-onderwysers, wat reeds etlike jare *Delphi*-onderrig gegee het, was dit ’n nuwe sillabus wat onderrig moes word en ’n nuwe benadering tot die onderrig van *Delphi*. Hulle moes heeltyd tydens *Scratch*-onderrig in gedagte gehou het waarheen hulle op pad was en op die oorgang na *Delphi* gefokus het, maar dit het nie altyd gebeur nie.

> *Nee, ek doen dit glad nie. Ek bring glad nie die twee [Scratch en Delphi] bymekaar nie. Dit is dalk ’n fout, maar ek doen dit glad nie so nie. Seker omdat ek ’n te beperkte kennis van Delphi het. Dalk kan ons volgende jaar sê volgende jaar se graad 11’s gaan dalk vinniger leer, omdat ek dalk sal weet wat aangaan. Dit was maar verlede jaar vir my, met Scratch ook, ’n totale nuwe leerervaring gewees. (P2)*

Onderwysers se onsekerheid is ook onderskryf deur hulle verwysing na begrippe wat nie in *Scratch* voorkom nie, soos die gebruik van MOD, SQRT en karakterhanteringsfunksies.

> *Ek dink dit is dat niks van hierdie begrippe in Scratch was nie. Scratch was eenvoudig gewees in die sin van dat daar nie ’n verskil tussen ’n String en ’n Integer was nie en nou begin hulle om ’n String te skep. Ek dink dat daardie, om ’n deel van ’n string af te sny en ’n nuwe een insit, nie daar was nie – ek weet nie. Ek dink daar was nie genoeg tyd gewees nie, of ek het dalk nie genoeg tyd met Scratch spandeer dat hulle dit kan visualiseer nie (P8).*

Soos reeds genoem (sien 4.3.2.2), het onderwysers later in hulle vertelling agtergekom dat begrippe soos MOD en karakterhantering wel in *Scratch* teenwoordig was.

Behalwe dat onderwysers onbevoeg gevoel het, het hulle ook gevoel dat hulle enigste bron van ondersteuning die graad 10 en graad 11 IT-handboeke was, wat hulle eers laat in die jaar ontvang het.

4.4.2 Die ontoereikendheid van handboeke

Die handboekkwessie het, tydens die tweede onderhoude, net soos met die eerste onderhoude (sien 4.2.1), die skuld gekry vir probleme. Volgens onderwysers het hulle sekere begrippe nie in *Scratch* onderrig nie, omdat dit nie in die handboek was nie.

> *Ek dink nie dit was in die Scratch-handboek nie (P8).*

Die graad 11 *Delphi*-handboeke was ook slegs in Engels beskikbaar en dit was veral vir Afrikaanse onderwysers 'n struikelblok.

> *Die Delphi is net in Engels, wat nog 'n probleem skep. Die benaming in die handboek is ook anders as waaraan ek gewoond is (P8).*

Alhoewel deelnemer P8 in die eerste onderhoude aangedui het dat die handboek nie slaafs gebruik word by *Scratch*-onderrig nie, was dit nou anders by *Delphi* en het P8 ook, net soos ander deelnemers, swaar op die inhoud van die handboek gesteun.

> *Basies sit ons en ek gaan deur die handboek (P8).*
>
> *Ek volg maar net wat die handboek doen (P2).*

Sommige onderwysers het verkies om terug te keer na die vorige handboek wat volgens die vorige sillabus by die onderrig van *Delphi* gebruik is.

> *Ek was een van 'n paar wat dadelik begin het met die nuwe handboek, maar ek het teruggegaan na die blou handboek [vorige handboek] toe. Ek steun baie sterk op die ou handboek vir daardie konkrete, vir daardie standvastigheid (P7).*

'n Moontlike rede hiervoor is dat die veranderings vir onderwysers baie moeilik was en hulle wou eerder dit wat aan hulle bekend was, gebruik. Dit kon egter tot nuwe probleme lei, aangesien alles wat graad 11-leerders volgens die KABV moes afhandel (DBE, 2011:30–40), nie in die vorige handboek was nie en dié handboek ook nie 'n objekgeoriënteerde benadering gevolg het nie.

Onderwysers het, behalwe die handboekkwessie, ook leerders se akademiese vermoë vir probleme in *Delphi* geblameer.

4.4.3 Leerders se akademiese vermoëns

Sommige onderwysers het gevoel dat dit weens hulle akademiese vermoë is dat leerders met *Delphi* gesukkel het en het dit vergelyk met dié leerders se prestasie in Wiskunde.

> *Ek dink dit is omdat hulle nie weet hoe om wiskunde te doen nie. Al die kinders is nie ewe sterk in wiskunde nie en as hulle dan berekeninge sien, dan skrik hulle. Ek het 'n groep wat nie so akademies sterk is soos gewoonlik nie. Ek het absoluut gemiddelde, gemiddelde kinders en ek kan amper sê ondergemiddeld, onderpresteerders (P4).*

Daarmee saam is aangevoer dat *Scratch* te maklik was en dat leerders wat nie goed vaar in programmering nie, nie in graad 10 deur 'n siftingsproses gegaan het voordat hulle in graad 11 aan *Delphi*-programmering blootgestel is nie.

> *… dat leerders dalk, jy weet, [voorheen] "gesif" was toe hulle in graad 10 Delphi gedoen het, dat hulle besef het dit is vir hulle te moeilik. Leerders het eintlik 'n magiese aantrekkingskrag tot Scratch, wat dit vir hulle maklik maak, en dan kom die regte programmeerdenke in graad 11 (P4).*

Daar is ook gesê dat, omdat *Scratch* so maklik was, leerders nie nodig gehad het om op programmeringsbegrippe te fokus en dit aan te leer nie.

> *Ek bedoel dit was vir hulle maklik gewees om die begrippe in Scratch te mis. Hulle het nie die begrippe nodig gehad om dit [programmering] in Scratch te doen nie (P8).*

Dit was opvallend dat die redes wat aangevoer is vir leerders se probleme met *Delphi,* in baie min gevalle met *Scratch*-onderrig geassosieer is. Ook die oplossings wat onderwysers vir probleme voorgestel het, het slegs in enkele gevalle verwys na 'n groter fokus op die onderrig van programmeringsbeginsels in *Scratch*. Onderwysers het ook selde hulle onderrig–leerstrategieë onder die vergrootglas geplaas.

Vervolgens word die onderrig–leerstrategie wat tydens *Delphi*-onderrig gevolg is, bespreek.

4.4.4 Onderrig–leerstrategie gevolg by *Delphi*

'n Subtiele skuif het in die onderrig–leerstrategie vanaf *Scratch* na *Delphi* by deelnemende onderwysers plaasgevind. *Scratch* het ruimte gelaat vir 'n meer informele onderrig–leerstrategie en eksperimentering, en onderwysers het gepoog

om dit deel te maak van hul onderrig–leerstrategieë (sien 4.2.4). *Delphi*-onderrig was egter meer gedoen deur demonstrasie van programme aan leerders, soos in die onderstaande aanhaling van P1 aangedui. Dit kan toegeskryf word aan die aard van *Delphi* (sien 2.5.2) en veral die spanning waaronder die deelnemende onderwysers was om die sillabus af te handel (sien 4.3.1.1 en 4.3.1.3).

> *Ek sal vir hulle 'n voorbeeld op die dataprojektor sit en dan sal ek vir hulle verduidelik. Dan verduidelik ek die program mondelings in gewone taal en as hulle nou almal verstaan, dan sal ek sê goed kom ons doen so 'n voorbeeld. Ek sal dit dan eers saam met hulle deurgaan en sê goed, doen dit, tik nou eers jou data in die tekslêer in, en dan sal ek sê goed, nou doen jy self een. Ons doen nou daardie voorbeeld of daardie oefening (P1).*

> *Ek het begin met baie verduidelikings, maar kinders sukkel baie met verduidelikings. Toe het ek maar oorgeskakel na demonstrasie toe waar ek vir hulle fisies demonstreer het en hulle daarna self 'n probleem moes oplos (P3).*

Daar was min geleenthede vir leerders om met *Delphi* te speel en te eksperimenteer, aangesien onderwysers beperkte tyd gehad het (4.3.1.3). Wanneer geleenthede wel geskep is, het dit goeie gevolge gehad.

> *Ek was heel verbaas met die RANDOM-funksie en die rondspeel met die muis en Button wat rondspring op die skerm, en toe begin 'n paar van hulle hulle eie virus-programmetjies te skryf. Na hulle laaste sessie het ons een middag gesit en toe hulle sien hoe alles in 'n groot program bymekaarkom, toe het hulle agtergekom dat daar 'n doel is met al die klein goedjies [streng sintaks] en het hulle begin positief raak oor Delphi (P3).*

Die gevolgtrekking word gemaak dat leerders die speelsheid van *Scratch* gemis het. Die min geleenthede vir eksperimentering en selfontdekking, kon bygedra het tot probleme met *Delphi*-onderrig.

Onderwysers het ook voorstelle gehad vir die oplos van probleme met *Delphi*, soos wat vervolgens bespreek word.

4.5 VOORSTELLE VAN ONDERWYSERS OM PROBLEME OP TE LOS

Kodes wat hierdie kategorie ingesluit het, is tyd wat aan *Scratch*-onderrig bestee word, vroeëre blootstelling aan *Delphi* en spesifieke onderrig van programmerings-begrippe in *Scratch*.

Aangesien die groot werkslading en die kwessie van tyd die onderwysers se grootste probleme was, het die meeste voorstelle vir oplossings daarvan verband gehou met ŉ besparing van tyd tydens *Scratch*-onderrig aan graad 10 leerders sodat vroeër met *Delphi* begin kon word.

> *Ek probeer hulle nou bietjie vinniger druk, om klaar te kry met Scratch, sodat ek in graad 10 bietjie met Delphi kan begin (P7).*

Nog ŉ voorstel van onderwysers om vroeër blootstelling aan *Delphi* te kry, was weer eens die vooruitverwysing na *Delphi*.

> *Ek sal baie Delphi in graad 10 tussendeur die Scratch inbring. Ek sal sê, in Delphi kry ons, en dan skryf ek sommer Delphi se IF en ELSE neer, dan sê ek dis nou die IF en ELSE en dan skryf ek Delphi se formaat op die bord neer (P1).*

Deelnemer P6 het voorgestel dat praktiese *Delphi*-voorbeelde van teoretiese begrippe tydens teoretiese werk aan leerders in graad 10 gewys word. Op hierdie wyse kan leerders reeds in graad 10 blootstelling aan *Delphi* kry, om tyd in graad 11 te bespaar.

> *Ek spaar ontsaglik tyd met hierdie kontroles en dit vul die teoriehoofstuk baie aan, want hulle sien dit is nie net Windows nie. In Delphi wys ek hulle nou van "uses", "uses" Windows, Messages, Dialogs (P6).*

Deelnemer P6 het verder voorgestel dat die beste manier sou wees om reeds in graad 10 tydens die onderrig van *Scratch* meer op programmeringsbegrippe te fokus.

> *Ek sien daarna uit om volgende jaar baie meer te kan fokus op veranderlikes of lusbeheerveranderlikes – al hierdie probleme waarvan ons gepraat het in graad 11. My voorstel is – moenie in graad 11 terugverwys na Scratch toe nie, begin al in graad 10 [om meer op programmeringsbegrippe en -beginsels te fokus] (P6).*

Deelnemer P3, wat op die onderrig van programmeringsbeginsels en -begrippe in *Scratch* gefokus het, en sukses behaal het met die oordrag daarvan (sien 4.3.2.1), se voorstelle was op begrippe wat reeds in *Scratch* onderrig kan word, gefokus. Een gedagte was om deur middel van die name van veranderlikes leerders reeds op die verskillende soorte, *String* en *Integer*, attent te maak.

> *Ek leer hulle sommer al klaar met afkortings in Scratch, van veranderlikes. "S" vir String, "I" vir Integer, sodat hulle dit nou al klaar [in Scratch] weet (P3).*

Net so het deelnemer P4 ook voorgestel dat meer klem in graad 10 op programmeringsbegrippe in *Scratch* gelê moet word. Dié begrippe sluit ingeboude

funksies, soos ABS en ROUND, die reikwydte van veranderlikes en die gebruik van verskillende datatipes in.

> *Ek lê meer klem op byvoorbeeld ABS en ROUND. Ek het spesifiek klem gelê op "For this Sprite only" en "For all Sprites" (P4).*

Verder het P3 beoog om leerders se begrip van herhaling te verdiep. Leerders moes nie net telkens die *FOREVER*-lus gebruik nie, maar moes die beste lusstruktuur kon kies en gebruik.

> *By die REPEAT laat ek hulle nie die normale REPEAT eers gebruik nie; hulle moet eers REPEAT vir X aantal kere met 'n teller doen, en hulle moet REPEAT … UNTIL gebruik. Ek laat hulle amper nie eers FOREVER gebruik nie, tensy hulle UNTIL gebruik (P3).*

Onderwysers het verskeie redes aangevoer vir probleme wat met *Delphi*-onderrig ervaar is, waarvan nie almal as gevolg van *Scratch* was nie. Hulle het ook verskeie voorstelle gemaak vir die oplos van probleme. Die voorstelle was nie altyd in ooreenstemming met voorskrifte vir die KABV nie, maar sou volgens onderwysers lei tot 'n besparing van tyd tydens die onderrig van *Scratch* ten einde vroeër met *Delphi* te kan begin. Slegs enkele deelnemers het voorgestel dat daar sterker op programmeringsbeginsels in *Scratch*-onderrig gekonsentreer moet word om sodoende die oordrag na *Delphi* makliker te maak.

4.6 SAMEVATTING

In hierdie hoofstuk is aangedui dat daar leemtes was met betrekking tot die onderrig van programmeringsbeginsels en -begrippe in *Scratch* aan graad 10-leerders. Verskeie redes is vir hierdie leemtes verskaf en die meeste kan toegeskryf word aan onderwysers se onsekerheid oor die onderrig van 'n nuwe programmeertaal. Die deelnemende onderwysers het gerapporteer dat hulle blootgestel was aan 'n nuwe programmeringsomgewing met min hulpbronne. Onderrig was 'n kwessie van probeer en tref en as gevolg daarvan is programmeringsbeginsels en -begrippe in *Scratch* nie behoorlik onderrig nie. Gevolglik was *Delphi*-onderrig aan dieselfde leerders in graad 11 ook onder druk. Onderwysers was oorlaai met werk omdat programmeringsbeginsels en -begrippe nie na graad 11 oorgedra is nie en die tyd wat volgens die KABV aan *Delphi*-onderrig toegeken is, nie haalbaar was nie.

Alhoewel daar groot verskille is tussen *Scratch* en *Delphi*, en alle programmeringsbeginsels en -begrippe nie in *Scratch* hanteer kan word nie, is daar bevind dat oordrag, van algemene programmeringsbeginsels en –begrippe met behoorlike onderrig in *Scratch*, wel moontlik is. Slegs enkele onderwysers kon egter voorstelle gee om die oordrag van programmeringsbeginsels en -begrippe na *Delphi* deur middel van *Scratch*-onderrig te verhoog. Die meeste deelnemers se voorstelle was gerig op ’n verkorting van die tydperk van *Scratch*-onderrig. Laasgenoemde is egter nie in ooreenstemming met riglyne vervat in die KABV nie. Die oplossing lê by die onderrig van *Scratch* op so ’n manier dat maksimum oordrag van programmeringsbeginsels en -begrippe na *Delphi* plaasvind. Hierdie voorstelle word in die volgende hoofstuk bespreek.

HOOFSTUK 5:

BEVINDINGS, GEVOLGTREKKINGS EN AANBEVELINGS

5.1 INLEIDING

Die doel met hierdie navorsing was om te bepaal hoe *Scratch* as 'n visuele programmeertaal (VP) aan graad 10-leerders onderrig behoort te word, om die oorgang na *Delphi* as sintaksisgebaseerde objekgeoriënteerde taal (SOOP) te vergemaklik. Sewe subdoelwitte en navorsingsvrae is geïdentifiseer om hierdie doel te bereik (sien 1.3). In hierdie hoofstuk word die bevindings, gevolgtrekkings en aanbevelings aan die hand van die navorsingsdoelwitte en -vrae bespreek.

5.2 BEVINDINGS EN GEVOLGTREKKINGS MET BETREKKING TOT NAVORSINGSDOELWIT 1: DIE AARD VAN *SCRATCH* AS PROGRAMMEERTAAL EN DIE ONDERRIG DAARVAN

Ten einde die navorsingsdoelwit te bereik, moes bepaal word wat die aard van *Scratch* as programmeertaal is en hoe dit onderrig behoort te word. Eerstens word bevindings en gevolgtrekkings ten opsigte van die aard van *Scratch* as programmeertaal bespreek en tweedens hoe *Scratch* onderrig behoort te word.

5.2.1 Die aard van *Scratch* as programmeertaal

Scratch is nie 'n volwaardige objekgeoriënteerde programmeertaal (OOP) nie, maar sommige aspekte daarvan word wel in *Scratch* aangetref (sien 2.4.4.1). *Scratch* is egter wel 'n volwaardige programmeertaal waarmee programme van verskeie vlakke van kompleksiteit geskryf kan word.

Die aard van *Scratch* kan in drie kernbeginsels saamgevat word, naamlik leer deur te speel, betekenisvol en sosiaal (sien 2.4.2). Die kleurvolle, animasieryke omgewing (sien 2.4.3 en figuur 2.2) moedig leerders aan om op 'n speelse wyse te leer programmeer, terwyl hulle projekte ontwikkel wat betekenisvol en interessant is (sien 2.4.2). Deel van die genot hiervan behels ook die uitruil van projekte en idees met ander *Scratchers* (sien 2.4.2).

Tydens die ontwerp van *Scratch* is ’n beleid van lae vloer, wye mure en hoë dak toegepas (Malan & Leitner, 2007:223) (sien 2.4.2). Wat die lae vloer-beleid betref, was dit belangrik dat leerders met die intrapslag ’n werkende program kan skryf, sonder moeilike kwessies ten opsigte van die geïntegreerde ontwikkelingsomgewing (GOO) (sien 2.4.3), sintaksis (sien 2.4.4.3), fouthantering (sien 2.4.5.3), uitvoering (sien 2.4.4.6) en stoor van programme. Die GOO van *Scratch* herinner aan ’n toneelopvoering, waar verskillende karakters (*Sprites*) rondbeweeg en aksies uitvoer wanneer gebruikers daarop klik. Programmering behels die insleep van gekleurde blokke kode wat soos legkaarte inmekaar pas, sodat sintaksisfoute uitgeskakel word (sien 2.4.4.3).

Alhoewel die lae vloer-beleid sowel as bogenoemde drie kernbeginsels daartoe kan lei dat *Scratch* leerders dalk aan ’n speletjie kan herinner, was die ontwerpers daarvan ook getrou aan die beleid van wye mure en ’n hoë dak (Malan & Leitner, 2007:223). Dit beteken dat *Scratch* ’n veeldoelige programmeertaal is waarmee komplekse programme vir ’n verskeidenheid toepassings ontwikkel kan word (sien 2.4.2).

In *Scratch* word begrippe wat in ander programmeertale abstrak is, konkreet en visueel voorgestel. Herhaling- en besluitnemingstrukture, instruksies en programmeringskode word kleurvol en met verskillende soorte blokke aangedui (sien figuur 2.3). Veranderlikes, sowel as die inhoud daarvan word ook konkreet voorgestel en geen reëls bestaan vir die benoeming van veranderlikes nie (sien 2.4.4.4). Geen datatipes word verklaar nie (sien 2.4.4.4), en omskakeling tussen datatipes word nie gedoen nie. Die vorms van parameter-gleuwe dui egter wel op verskillende datatipes, naamlik getalle, teks en Boole-waardes (sien tabel 2.3).

Uitvoering van programme kan stap-vir-stap waargeneem word, en opgestel word om in stadige aksie uit te voer, vir opsporing van logiese foute (sien 2.4.4.6).

Opsommend kan *Scratch* dus beskryf word as ’n visuele programmeertaal met ’n speelse aard, wat programmeringsbegrippe konkreet voorstel. Die animasieryke omgewing moedig leerders aan om met programmering te eksperimenteer. Kwessies wat programmering gewoonlik kompliseer, soos sintaksis en onvriendelike

foutboodskappe, is uitgeskakel en ’n wye verskeidenheid toepassings kan daarmee ontwikkel word.

Vervolgens word bevindings en gevolgtrekkings rakende die onderrig van *Scratch* bespreek.

5.2.2 Die onderrig van *Scratch*

Vir die hedendaagse leerder is *Scratch* die ideale omgewing vir ’n eerste kennismaking met programmering. *Scratch* verskaf ’n platform vir die onderrig van programmering met minder struikelblokke en meer positiewe ervarings (Kapsimali & Sampson, 2011:186; Malan & Leitner, 2007:223; Meerbaum-Salant *et al.*, 2013:239) (sien 2.8.2.1).

As gevolg van die boublok-benadering en aard van *Scratch (*sien 2.4.2) kan *Scratch* maklik aangeleer word (sien 2.4.4.2). Visuele aspekte van *Scratch* (sien 2.4) is ’n groot bate by die onderrig van programmering, en moet benut word om akkurate geheuemodelle van abstrakte programmeringsbegrippe te vorm (Jehng *et al*., 1999:288). Aangesien programinstruksies nie ingetik hoef te word nie (sien 2.4.4.2), kan onderrig van *Scratch* van die begin af op die ontwerp van oplossings en die logika van programmering fokus (sien 2.4.5.1). Tyd hoef nie aan sintaktiese reëls en die oorbeklemtoning daarvan afgestaan te word nie (Malan & Leitner, 2007:223), en is dus beskikbaar vir die onderrig van programmeringsbeginsels en -begrippe wat daarin vervat is (sien 2.4.4).

Vorige navorsing bevestig dat sekere programmeringsbegrippe onder bepaalde omstandighede wel deur *Scratch* aangeleer kan word (Malan & Leitner, 2007:223; Maloney *et al.,* 2010:7–13; Meerbaum-Salant *et al.,* 2013:263) (sien 2.8.1) en bevindings ten opsigte van die onderrig van *Scratch* word vervolgens bespreek.

Scratch behoort so onderrig te word dat leerders so gou moontlik kan begin programmeer (sien 2.8.1). *Scratch*-onderrig is ’n kombinasie van eksplorasie en spel, maar die fokus moet deurgaans wees op die aanleer van die kuns van programmering (DBE, 2011:21; Wolz *et al*., 2009:3) (sien 2.8.1).

Probleme uit die werklike lewe wat met bekende, opwindende situasies, toepaslik vir leerders se ervaringswêreld verband hou (DBE, 2011:21; Wang en Zhou, 2011:490), moet ontleed word. Oplossings vir hierdie probleme moet dan met pseudokode of algoritmes (sien 2.3) voorgestel word (DBE, 2011:21; Malan en Leitner 2007:225, Wang en Zhou, 2011:488), en bepaalde programmeringsbegrippe wat daarin voorkom, moet uitgewys word. Leerders kan ook gevra word om self probleemscenario's waarin bepaalde programmeringsbegrippe, soos deur die onderwyser uitgewys, voorkom, uit te dink. Voltooide *Scratch*-projekte moet deur leerders bestudeer word en hulle moet daarop voortbou. Sodoende sal leerders 'n aanduiding kry van hoe programmeerders dink en hoe om programmeringsprobleme te benader (Malan & Leitner (2007:225) (sien 2.8.1).

Opsommend kan dus gesê word dat die onderrig van *Scratch* gewoonlik op informele wyse plaasvind en op eksplorasie en spel staatmaak. Binne 'n formele onderrigsituasie, waar die doel is om programmeringsbeginsels en –begrippe aan te leer, moet hierdie eksplorasie en spel egter nie ongekontroleerd wees nie en verg dit goeie voorbereiding van die onderwyser sodat leerders deur ontdekking en spel tog basiese programmeringsbeginsels en –begrippe kan aanleer. Leerders sal baat vind indien hulle toepassings ontwikkel wat met hulle ervaringswêreld verband hou en hulle belangstelling prikkel. Die fokus moet egter altyd op die onderrig van probleemoplossing en programmeringsbegrippe wees om leerders 'n idee te gee van hoe programmeerders dink en probleme oplos.

Die bevindings en gevolgtrekkings ten opsigte van navorsingsdoelwit 2, naamlik die aard van *Delphi*, word vervolgens bespreek.

5.3 BEVINDINGS EN GEVOLGTREKKINGS MET BETREKKING TOT NAVORSINGSDOELWIT 2: DIE AARD VAN *DELPHI* AS PROGRAMMEERTAAL

Navorsingsdoelwit 2 het ten doel gehad om te bepaal wat die aard van *Delphi* as programmeertaal is (sien 1.3).

Delphi is 'n volwaardige programmeertaal waarmee komplekse toepassings ontwikkel kan word (sien 2.5.4). Die verwysing na *Delphi* as 'n SOOP beteken dat 'n

programmeerder self die programmeringskode volgens sintaktiese reëls van die taal moet intik en dat ŉ objekgeoriënteerde benadering gevolg word (sien 2.5). Alhoewel *Delphi* ŉ volwaardige objekgeoriënteerde programmeertaal is (sien 2.5.2), kan programmering ook vanuit ŉ prosedurele benadering gedoen word (sien 2.5.2 en 2.7.2).

Delphi voorsien ŉ grafiese gebruikerskoppelvlak, en oplossings word deur middel van die GOO geïmplementeer (sien figuur 2.6). Objekte word op ŉ vorm geplaas (sien 2.5.3) en programmeringskode word dan volgens die sintaktiese en semantiese reëls van die taal ingetik (sien 2.5.4.1 en 2.5.4.2). Hierdie programmeringskode moet na masjienkode vertaal (gekompileer) word om ŉ uitvoerbare oplossing te skep. ŉ *Delphi*-oplossing bestaan uit ŉ samestelling van verskeie lêers waarvan die stoor, in teenstelling met die enkele lêer van ŉ *Scratch*-oplossing, ŉ aanvanklike komplekse proses is (sien 2.5.3). Die gebruik van ŉ *Delphi*-program hou verband met bepaalde gebeurtenisse wat aan objekte gekoppel word, soos byvoorbeeld die klik van ŉ knoppie of die oopmaak van ŉ vorm (sien 2.5.3).

Om die aard van *Delphi* verder te omskryf, word bevindings rakende spesifieke aspekte van *Delphi*, naamlik probleemoplossing, veranderlikes en datatipes, herhaling- en besluitnemingstrukture, en fouthantering en toetsing vervolgens bespreek.

5.3.1 Probleemoplossing

Soos by ander programmeertale (sien 2.3 en 2.5.5.1) is probleemoplossing, wat ook die ontwerp van algoritmes behels, een van die belangrikste stappe by *Delphi*-programmering. Die intik van programmeringskode kan as ŉ implementering van algoritmes beskou word (sien 2.3), wat beteken dat die probleem op daardie stadium reeds opgelos behoort te wees en dat die oplossing slegs in *Delphi*-kode oorgetik moet word. Die ontwerp van ŉ gebruikerskoppelvlak sowel as die hantering van foutiewe toevoerdata deur die gebruiker maak ook deel uit van die probleemoplossingsproses in *Delphi* (sien 2.5.5.1).

5.3.2 Veranderlikes en datatipes

Veranderlikes is ŉ abstrakte begrip in *Delphi*, wat moeilik deur beginnerprogrammeerders begryp word (Maloney *et al.,* 2010:6; Meerbaum-Salant *et al*., 2013:240; Resnick *et al.,* 2009:65). Verskeie ander aspekte wat hiermee gepaardgaan, naamlik benoeming, toekenning van datatipes, omskakeling tussen getal- en tekstipes, en reikwydte dra by tot die kompleksiteit van die begrip van veranderlikes (sien 2.5.4.3).

5.3.3 Herhaling- en besluitnemingstrukture

Delphi bevat verskeie herhaling- en besluitnemingstrukture, waarvan elk ŉ spesifieke funksionaliteit, sintaksis en reëls vir die programmering daarvan het (sien 2.5.4.4). Ten einde die doeltreffendste herhalingstruktuur vir die oplossing van ŉ probleem te kies, moet die *Delphi*-programmeerder ŉ grondige begrip van die werking van elke herhalingstruktuur hê, sowel as van die probleem wat opgelos moet word. Net so bestaan daar verskeie besluitnemingstrukture in *Delphi* (sien 2.5.4.4). Ten einde besluitnemingstrukture te programmeer, word Boole-voorwaardes benodig en deeglike kennis van die nodige simbole, sintaksis, operators en die logiese werking daarvan word vereis (sien 2.5.4.4).

5.3.4 Fouthantering en toetsing

Verskeie soorte foute moet deurgaans deur die programmeerder gehanteer word, naamlik sintaksisfoute, logiese foute en uitvoerfoute (sien 2.5.5.3). Indien ŉ sintaksisfout gemaak is, word dit met ŉ rooi-gemerkte boodskap aangedui, wat gewoonlik nie gebruikersvriendelik is nie (sien 2.5.4.2). *Delphi*-programmeerders moet deurgaans in hulle ontwerp en implementering van oplossings voorsiening maak vir die voorkoming van bogenoemde foute. Dit impliseer dat hulle ook moet antisipeer watter foutiewe waardes gebruikers moontlik kan insleutel, wat kan veroorsaak dat werkende programme staak (sien 2.5.5.3). Fouthantering en deeglike toetsing van programme vereis dus baie deeglike beplanning en deursettingsvermoë van die *Delphi*-programmeerder (sien 2.3 en 2.5.5.3).

Om navorsingsdoelwit 2 te beantwoord, is bevindings en gevolgtrekkings rakende die aard van *Delphi* (sien 2.5.2) bespreek. Hieruit is dit duidelik dat *Delphi* 'n komplekse programmeertaal is wat gespesialiseerde kennis van die gebruiker vereis.

Ná die bespreking van die aard van *Scratch* en *Delphi* word bevindings en gevolgtrekkings met betrekking tot navorsingsdoelwit 3 bespreek.

5.4 BEVINDINGS EN GEVOLGTREKKINGS MET BETREKKING TOT NAVORSINGSDOELWIT 3: ALGEMENE PROGRAMMERINGSBEGINSELS WAT TYDENS *SCRATCH*-ONDERRIG VASGELÊ KAN WORD

Die volgende doelwit wat gestel is, was om te bepaal watter algemene beginsels van programmering reeds by die onderrig van *Scratch* vasgelê kan word. Uit die literatuur is bevind dat *Scratch* bykans al die basiese programmeringsbeginsels en -begrippe bevat wat in ander programmeertale gebruik word (sien 2.4.4).

Scratch-programmeerders moet steeds deur al die stappe in die proses van programmering werk, naamlik probleemoplossing, ontwerp, implementering en instandhouding (sien figuur 2.1). Die *Scratch*-programmeerder moet verder ook vertroud wees met die ontwerp van 'n GOO, die begrip van veranderlikes, herhaling- en besluitnemingstrukture, logiese volgorde van instruksies, programuitvoering en fouthantering en toetsing (sien 2.4.4).

Daar is wel gevorderde programmeringsbeginsels en -begrippe wat nie in *Scratch* voorkom nie, naamlik oorerflikheid, polimorfisme, abstrakte datatipes en metodes (sien 2.4.4.1). Alhoewel *Scratch* nie as 'n volwaardige OOP gesien word nie, word sommige aspekte van objekte wel in *Scratch* aangetref (sien 2.4.4.1 en tabel 2.4).

Wanneer die taalkwessies, soos in die boonste paragrawe bespreek, opsygeskuif word en die klem op die aanleer van programmeringsbeginsels en die proses van programmering geplaas word, is die ooreenkomste tussen programmering in *Scratch* en *Delphi* merkwaardig (sien tabel 2.4). Word die tale vergelyk ten opsigte van die vyf domeine van programmering wat Du Boulay (1989) geïdentifiseer het (verwysing in Sajaniemi & Kuittinen 2008:76; Bennedsen & Caspersen, 2008:9) (sien 2.3), verskil die tale slegs ten opsigte van notasie (Kelleher en Pausch, 2005:83), naamlik

die programmeringskode wat ingetik moet word. Programmeerders in beide tale het dieselfde doelwit, naamlik om algoritmes (sien 2.3) te ontwikkel om probleme op te los en ’n reeks simbole vir die doel te manipuleer (Dijkstra, 1989:1392; Kelleher & Pausch, 2005:83).

Scratch verskaf dus ’n stewige grondslag waarop leerders kan bou vir die oorgang na ander programmeertale. (Malan & Leitner, 2007:223; Maloney *et al*., 2010:7–13; Meerbaum-Salant *et al*., 2013:263). Die gevolgtrekking word dus gemaak dat probleemoplossing, ontwerp, implementering, instandhouding sowel as sommige aspekte van die notasie van *Delphi* reeds in *Scratch* aangeleer kan word. Die vraag ontstaan egter hoe die algemene beginsels van programmering tydens die onderrig van *Scratch* onderrig behoort te word met die oog op die latere bemeestering van *Delphi.*

5.5 BEVINDINGS EN GEVOLGTREKKINGS MET BETREKKING TOT NAVORSINGSDOELWIT 4: ONDERRIG VAN ALGEMENE PROGRAMMERINGSBEGINSELS IN *SCRATCH*

Navorsingsdoelwit 4 is spesifiek gerig op die onderrig van *Scratch* met die oog op latere bemeestering van *Delphi*. In dié verband sal *Scratch* noodwendig anders onderrig word, aangesien dit nie net daarom gaan om ’n liefde vir programmering te kweek nie (sien 2.4.1), maar om die latere bemeestering van *Delphi*. Die kennis en vaardighede wat graad 10-leerders met *Scratch* opdoen, moet verband hou met en toegepas kan word in *Delphi.* Die oorgang na *Delphi* behoort nie ’n radikale nuwe ervaring te wees nie. *Scratch*-onderrig moet leerders op programmering voorberei sodat hulle, sonder die afleiding van sintaksis, op die prosesse van programmering en programmeringsbegrippe kan konsentreer (Kelleher & Pausch, 2005:131).

5.5.1 Eksplisiete onderrig van programmeringsbegrippe

Navorsing van Carver (1986:12) en Meerbaum-Salant *et al*. (2013:263) het aangetoon dat, wanneer die doel is om die oordrag na ’n ander programmeertaal te vergemaklik, programmeringsbegrippe eksplisiet aan leerders onderrig moet word (sien 2.8.1). Leerders kan nie op hulle eie gelaat word om teen hulle eie tempo aan projekte van hulle keuse te werk en begrippe te ontdek nie (Maloney *et al.,* 2008:370;

Meerbaum-Salant *et al*., 2013:263) (sien 2.8.1), maar benodig bekwame leiding van onderwysers vir die bemeestering van fundamentele programmeringsbeginsels en -begrippe. Indien onderrig op selfontdekking staatmaak, bestaan die gevaar dat swak programmeringsbeginsels aangeleer kan word en dat leerders nie die diepere werking van byvoorbeeld herhalingstrukture begryp nie (sien 2.8.2.2).

Onderrig moet dus so beplan word dat spesifieke programmeringsbeginsels en -begrippe onderrig word, bewustheid daarvan by leerders gekweek word, en nadenke van oplossings aangemoedig word. Indien programmeringsbegrippe nie deur onderwysers by die naam genoem word nie, kan dit veroorsaak dat leerders nie daarvan bewus is nie (Meerbaum-Salant *et al*., 2013:244) (sien 2.8.1). Leerders se aandag kan ook maklik afgelei word deur die visuele materiaal van *Scratch* en die klem moet dus op die aanleer van goeie programmeringsbeginsels geplaas word, eerder as op spesifieke *Scratch*-eienskappe soos animasie (sien 2.8.1).

Elke moontlike begrip wat in *Scratch* onderrig kan word, moet met erns ten opsigte van die oordrag daarvan na *Delphi* bejeën word. Onderwysers moet vanuit ’n *Delphi*-oogpunt na *Scratch* kyk, sodanige begrippe identifiseer en elke moontlike geleentheid gebruik om dit te onderrig.

Wiskundige funksies, soos ROUND, RANDOM, SQRT, ABS en die operator MOD is maar enkele voorbeelde van nog verdere programmeringsbegrippe wat met *Scratch*-onderrig kan word (sien tabel 2.4). Ook vaardighede om stringe deur middel van bepaalde funksies en operators te manipuleer, kan reeds met *Scratch*-onderrig gevestig word. *Scratch*-onderrig bied ’n gulde geleentheid om aan leerders te verduidelik wat ’n funksie is en hulle aan begrippe met betrekking tot funksies, soos roepinstruksies en parameters, bekend te stel om sodoende ’n stewige grondslag te verskaf vir die skryf van funksies in *Delphi*.

5.5.2 Eksplisiete vaslegging van objekgeoriënteerde programmerings-beginsels en -begrippe

Alhoewel *Scratch* nie ’n volwaardige OOP is nie, is daar wel bepaalde objekgeoriënteerde begrippe wat reeds in *Scratch* gevestig kan word (sien 2.4.4.1) en onderwysers behoort dit so vroeg moontlik aan leerders voor te hou. Die karakters in die mikrowêreld van *Scratch*, kan byvoorbeeld gebruik word om die begrip van

objekte konkreet aan leerders voor te hou. Dit is belangrik dat 'n objekgeoriënteerde denkwyse (Weisfeld, 2009:1) reeds met *Scratch*-onderrig gekweek word. Onderrig in *Delphi*, as volwaardige SOOP, moet dan slegs hierop voortbou sodat programmeringsbeginsels en -begrippe nie van nuuts af onderrig word nie.

In tabel 5.1 word 'n sintese gegee van die bevindings en gevolgtrekkings van navorsingsdoelwit 4 en in tabel 5.2 van die aanbevelings ten opsigte hiervan.

Tabel 5.1: Bevindings en gevolgtrekkings van navorsingsdoelwit 4

- Onderwysers moet *Scratch* onderrig met die oog op die oorgang na *Delphi*.
- Verwys na objekgeoriënteerde begrippe en stel terminologieë aan leerders bekend deur die verwysing na *Sprites* as objekte.
- *Scratch* moet nie aan selfontdekking alleen oorgelaat word nie.
- Tydens die beplanning van *Scratch*-onderrig moet oor die onderrig van spesifieke programmeringsbegrippe gereflekteer word.
- Leerders moet spesifieke ondersteuning ontvang ten opsigte van programmeringsbeginsels en –begrippe wat nie noodwendig in *Scratch* begripvorming vereis nie.
- Noem programmeringsbegrippe by die naam om leerders bewus te maak van terminologieë.
- Vra leerders om self probleemscenario's wat bepaalde programmeringsbegrippe bevat, uit te dink.
- Stel oplossings van probleme altyd deur middel van algoritmes en/of pseudokode voor en wys programmeringsbegrippe wat daarin vervat is aan leerders uit.
- Probleme waarvoor oplossings ontwikkel moet word, moet verband hou met opwindende probleme uit die werklike lewe wat op tieners van toepassing is.
- Voltooide projekte moet deur leerders bestudeer word, waarna hulle daarop moet voortbou.

In die voorafgaande bespreking, is bevindings en gevolgtrekkings ten opsigte van navorsingsdoelwitte 1 tot 4 aan die hand van die literatuur wat bestudeer is, bespreek. Die bespreking van die bevindings van navorsingsdoelwitte 5 en 6 het betrekking op die bevindings en gevolgtrekkings van die empiriese ondersoek.

5.6 BEVINDINGS EN GEVOLGTREKKINGS MET BETREKKING TOT NAVORSINGSDOELWIT 5: HUIDIGE *SCRATCH*-ONDERRIG DEUR ONDERWYSERS

Onderwysers het *Scratch* in 2012 die eerste keer aan graad 10 IT-leerders onderrig (sien 3.5.2). Daar is bevind dat programmeringsbeginsels en -begrippe nie pertinent onderrig is nie en dat onderwysers meestal op leerders se intuïtiewe begrip van programmeringsbegrippe in *Scratch* staatgemaak het (sien 4.2.3.6). Onderwysers was ook onseker oor hoe om *Scratch* te onderrig (sien 4.2.1). Die taal van programmering is nie gepraat nie – programmeringsbegrippe is nie by name genoem nie en is eers in graad 11 toe *Delphi* gebruik is, aan leerders bekend gestel. Dieselfde is ook deur Meerbaum-Salant *et al*. (2013:243) in hulle navorsing bevind (sien 2.8.1). Waardevolle tyd het gevolglik tydens die onderrig van programmeringsbeginsels en -begrippe verlore gegaan (sien 4.3.1.1 en 4.3.13) aangesien *Scratch*, in plaas van programmering in *Scratch* onderrig is, en programmeringsbegrippe vir latere onderrig tydens *Delphi* in graad 11 uitgestel is.

Onderwysers het die bevordering van logiese denke en die aanleer van probleemoplossing as die belangrikste aspekte van *Scratch*-onderrig gesien en het baie aandag aan die onderrig hiervan gegee. Hulle was egter onseker hoe om dit by die onderrig van *Scratch* te integreer. Die onsekerheid was spesifiek met betrekking tot die hoeveelheid tyd wat aan algoritme-ontwerp bestee moes word. Onderwysers het gevolglik nie algoritme-ontwerp tydens elke *Scratch*-program gedoen nie, maar dit meestal as aparte onderwerp hanteer. In baie gevalle het leerders dit dus nie as ’n noodsaaklike stap in enige programmeringsopdrag beskou nie (sien 4.2.2).

Terwyl algoritme-ontwerp en probleemoplossing vir onderwysers belangrik was, is ander programmeringsbegrippe skynbaar verwaarloos. Geen melding is gemaak van foutopsporing of toetsing van *Scratch*-programme nie. Slegs sommige aspekte van die eerste drie stappe van die proses van programmering (sien figuur 2.1), naamlik probleemvoorstelling, ontwerp en implementering, is onderrig. Vaardighede soos die opspoor van foute en kritiese refleksie oor oplossings (sien figuur 2.9) is nie onderrig nie (sien 4.2.2 en 4.2.3.6).

Aspekte ten opsigte van datatipes en veranderlikes wat met *Scratch* onderrig kan word, is slegs deur enkele onderwysers geïdentifiseer en oppervlakkig aan leerders verduidelik. Onderwysers het meestal op leerders se intuïtiewe begrip van veranderlikes staatgemaak, as gevolg van *Scratch* se konkrete, visuele voorstelling daarvan. Geen melding is gemaak van reikwydte van veranderlikes of benoemingskonvensies nie (sien 4.2.3.3) en slegs enkele verwysings na datatipes is gemaak (sien 4.2.3.3).

Ook die begrip van herhalingstrukture is meestal aan selfontdekking oorgelaat. Onderrig was nie spesifiek op die onderskeid tussen voorwaardelike en onvoorwaardelike herhalingstrukture gerig nie. Leerders het meestal herhalingstrukture van hulle keuse, veral FOREVER, gebruik om probleme op te los. In plaas daarvan om die algemene beginsels van herhalingstrukture in *Scratch* te bespreek, het onderwysers vooruitverwys na herhalingstrukture in *Delphi* (sien 4.2.3.1).

Onderrig ten opsigte van besluitnemingstrukture was beperk tot eenvoudige besluitneming tussen twee keuses. Geen melding is gemaak van die onderrig van geneste keusestrukture of Boole-uitdrukkings nie. Die gevolgtrekking word dus gemaak dat onderwysers op leerders se intuïtiewe begrip rakende besluitneming in *Scratch* staatgemaak het (sien 4.2.3.2).

Verskeie ander programmeringsbeginsels en -begrippe is ook nie pertinent in *Scratch* onderrig nie. *Scratch* is deur die meeste onderwysers nie as ’n programmeringsomgewing beskou ten einde objekgeoriënteerde begrippe aan leerders te onderrig nie (sien 4.2.3.4). Geen melding is gemaak van die onderrig van lyste in *Scratch* nie. Een onderwyser het wel na tekslêers verwys, maar het dit weer eens deur middel van ’n vooruitverwysing na *Delphi* aan leerders probeer verduidelik (sien 4.2.3.5).

Onderwysers het oor die algemeen ’n tradisionele, direkte onderrig–leerstrategie met die onderrig van *Scratch* toegepas, maar het dit ook gekombineer met informele onderrig–leerstrategieë soos eksplorasie en demonstrasie (sien 4.2.4). Sommige onderwysers het egter hul kommer uitgespreek dat alternatiewe onderrig–leerstrategieë klasdissipline sou benadeel. Pogings om alternatiewe onderrig–

leerstrategieë toe te pas, was gevolglik steeds verweef met die tradisionele, direkte onderrig–leerstrategie (sien 4.2.4).

'n Ander belangrike kwessie vir onderwysers, was die hoeveelheid tyd wat aan *Scratch*-onderrig bestee moet word. In 2012 is die volle jaar aan *Scratch*-onderrig bestee. Onderwysers was egter bekommerd oor die tyd wat vir *Delphi*-onderrig in graad 11 en graad 12 toegeken is, en dat leerders nie al die programmeringsbeginsels en -begrippe sou bemeester nie. Volgens hulle was 'n moontlike oplossing om reeds in graad 10 met *Delphi*-onderrig te begin en minder tyd aan *Scratch* te bestee (sien 4.2.5).

Die gevolgtrekking word gemaak dat onderwysers *Scratch* sien as 'n middel om probleemoplossing en logiese denke te onderrig, maar nie as 'n programmeertaal om ander programmeringsbegrippe sowel as die programmeringsproses te onderrig nie (sien 4.2.3.6). *Scratch*-onderrig, soos dit in 2012 in die skole waarbinne hierdie navorsing uitgevoer is geïmplementeer is, maak baie staat op leerders se intuïtiewe begrip van programmeringsbeginsels en -begrippe en word nie deur goeie onderrig–leerstrategieë gerugsteun nie. Om onderwysers in staat te stel om genoeg tyd te hê om programmeringsbeginsels en -begrippe aan leerders te onderrig, het hulle besluit om eerder minder tyd aan *Scratch* te bestee, sodat meer tyd aan die onderrig van *Delphi*-programmering gewy kan word.

Vervolgens word die bevindings en gevolgtrekkings van die empiriese ondersoek, met betrekking tot navorsingsdoelwit 6 bespreek.

5.7 BEVINDINGS EN GEVOLGTREKKINGS MET BETREKKING TOT NAVORSINGSDOELWIT 6: HOE ONDERWYSERS DIE OORGANG VANAF DIE ONDERRIG VAN *SCRATCH* NA DIE ONDERRIG VAN *DELPHI* ERVAAR HET

Onderwysers se waarneming was dat sommige programmeringsbeginsels en -begrippe wel na *Delphi* oorgedra is (sien 4.3.2.1), maar nie in die mate wat daar in die KABV (sien addendum F) van hulle verwag is nie. Die oordrag wat plaasgevind het, kan as 'n basiese bewusmaking van programmeringsbeginsels en –begrippe gesien word en was nie genoegsaam om met gevorderde *Delphi*-programmering te

begin nie. Dit is waarom onderwysers ’n tydsprobleem ondervind het en baie programmeringsbeginsels en –begrippe weer moes onderrig. Hulle grootste bekommernis was die tempo waarteen die sillabus afgehandel moes word (sien 4.3.1.1 en 4.3.1.3).

Die *Delphi*-GOO was aanvanklik vir leerders vreemd, maar hulle het wel mettertyd die werking daarvan aangeleer (sien 4.3.1.2). Dit het egter meer tyd geneem om die *Delphi*-GOO aan leerders bekend te stel, as waarvoor in die KABV voorsiening gemaak is (sien 4.3.1.3) en onderwysers was gevolglik agter met die beplande onderrig. Dit het gelei tot verdere spanning oor die groot volume werk wat in graad 11 afgehandel moes word. Die KABV se werkswyse het aangeneem dat bepaalde programmeringsbeginsels en -begrippe reeds in graad 10 met *Scratch* aangeleer is (sien addendum A), maar onderwysers het gevoel dat hulle in graad 11 van voor af moet begin met die onderrig van hierdie programmeringsbeginsels en -begrippe (sien 4.3.1.1 en 4.3.1.3).

Die sintaksis van *Delphi* was aanvanklik vir leerders vreemd en het ook tyd geneem om aan te leer. Onderwysers het algemene foute rakende sintaksis, soos die weglating van aanhalingstekens en kommapunte, en die weglating van BEGIN- en END-instruksies in programstrukture nie as struikelblokke gesien nie, maar as die normale verloop van die aanleer van ’n sintaksisgebaseerde programmeertaal (sien 4.3.1.2).

Fouthantering was egter leerders se grootste struikelblok. Leerders is nie in *Scratch* daaraan blootgestel nie en het dit gevolglik moeilik gevind om foutopsporing in *Delphi* te doen. Leerders kon wel mettertyd sintaksisfoute opspoor, maar hulle het steeds gesukkel om logiese foute op te spoor (4.3.1.2).

Ten spyte van bogenoemde probleme was sommige onderwysers ook van mening dat *Scratch* leerders gehelp het om *Delphi* vinniger te verstaan (sien 4.3.2.1). Leerders het daarin geslaag om hulle logiese denke en probleemoplossingsvaardighede na *Delphi* oor te dra. Onderwysers wat nie algoritmes in graad 10 onderrig het nie, het egter aangedui dat dit in graad 11 vir hierdie leerders problematies was (sien 4.3.2.1). Net so het die onderwyser wat basiese begrippe ten opsigte van objekte reeds in *Scratch* onderrig het, ervaar dat

graad 11-leerders ’n beter begrip van objekte sowel as die eienskappe daarvan gehad het, teenoor leerders wat nie voorheen in *Scratch* rakende objekte onderrig is nie (sien 4.3.2.1). Leerders kon ook die begrip van tekslêers na *Delphi* oordra, alhoewel hulle dit steeds moeilik gevind het om dit in *Delphi* toe te pas (sien 4.3.2.1).

Verskeie basiese programmeringsbegrippe is dus volgens onderwysers se ervaring na *Delphi* oorgedra, maar komplekse toepassings van hierdie begrippe sowel as ’n diepere begrip daarvan het ontbreek (sien 4.3.2.1). Leerders het byvoorbeeld herhaling verstaan, maar het gesukkel om die werking van lusbeheerveranderlikes en die verskil tussen voorwaardelike en onvoorwaardelike herhalingstrukture te verstaan.

Onderwysers het erken dat hulle nie in hulle beplanning van *Scratch*-onderrig vir onderrig van programmeringsbegrippe met die oog op *Delphi* voorsiening gemaak het nie. Hulle het ook nie op spesifieke begrippe in *Scratch*, soos byvoorbeeld die bewerkingsoperator MOD, funksies soos byvoorbeeld SQRT en karakterhanterings-funksies en Boole-operators klem gelê nie, en het selfs getwyfel of hierdie begrippe in *Scratch* voorkom (sien 4.3.2.2). Hierdie begrippe is gevolglik nie na *Delphi* oorgedra nie en moes dus weer in graad 11 onderrig word.

Leerders kon wel tydens *Delphi*-onderrig terugwaartse assosiasies na *Scratch* maak en daarop voortbou. In een geval het leerders selfs op hulle eie assosiasies gemaak sonder dat die onderwyser daaraan gedink het, en het hulle hul onderwyser daarop gewys (sien 4.3.2.1).

Aangesien onderwysers nie die graad 10 jaar gesien het as ’n jaar waarin programmering onderrig word nie, het hulle vasgebrand met onderrigtyd in graad 11 (sien 4.3.1.1 en 4.3.1.3). Die werkslading was nou soveel groter – *Delphi* sowel as programmeringsbegrippe en -beginsels wat reeds in graad 10 gevestig moes wees (sien addendum F), moes nou in graad 11 onderrig word.

Indien programmeringsbeginsels en -begrippe dus nie op behoorlike wyse aan graad 10-leerders in *Scratch* onderrig word nie, sal ’n beperkte, of selfs geen begrip daarvan na *Delphi* oorgedra word, soos wat in hierdie geval onderwysers se ervaring was. In gevalle waar programmeringsbeginsels en -begrippe egter behoorlik in

Scratch onderrig is, het onderwysers ervaar dat goeie begrip daarvan na *Delphi* oorgedra is (sien 4.3.2.1).

Daar was ook ander bevindings en gevolgtrekkings met betrekking tot die onderrig van *Scratch* en die oorgang na *Delphi*, wat nie direk op die navorsingsdoelwitte betrekking gehad het nie, wat vervolgens bespreek word.

5.8 ANDER BEVINDINGS EN GEVOLGTREKKINGS

Ander bevindings en gevolgtrekkings wat nie direk van toepassing is op die navorsingsdoelwitte nie (sien 1.3), het ook uit die empiriese ondersoek na vore gekom. Dit is egter steeds relevant met betrekking tot die onderrig van *Scratch* en die oorgang na *Delphi* en word dus ook bespreek.

Scratch-onderrig was vir onderwysers ’n nuwe leerervaring. Al was van hulle ervare IT-onderwysers, het *Scratch*-onderrig ’n nuwe onderrig–leerbenadering vereis en onderwysers was nie seker hoe om *Scratch* te onderrig nie en nog minder hoe om dit te onderrig met die oog op die oorgang na *Delphi* (sien 4.4.1).

5.8.1 Handboeke

Onderwysers het swaar op handboeke gesteun vir riglyne rakende die onderrig van *Scratch*, maar handboeke was om verskeie redes aanvanklik nie beskikbaar nie. Sekere begrippe is nie met behulp van *Scratch* onderrig nie, omdat dit, volgens onderwysers, nie in die handboeke was nie. Vir die onderrig van *Delphi* het sommige onderwysers verkies om die vorige IT-handboeke, wat vir die vorige NKV-sillabus geskryf was, te gebruik.

5.8.2 Onderrig–leerstrategieë

Onderwysers het nie die probleme wat met *Delphi*-onderrig ervaar is aan hulle eie onderrig–leerstrategieë toegeskryf nie of oor hulle *Scratch*-onderrig gereflekteer nie (sien 4.4). Hulle het eerder die probleme aan ander faktore toegeskryf, soos leerders se akademiese vermoëns en die handboekkwessie (sien 4.4.3 en 4.4.2). In *Scratch* word leerders toegelaat om te eksperimenteer, en onderrig is gewoonlik redelik informeel (sien 4.2.4), maar *Delphi*-onderrig is meer voorskriftelik, formeel en

tradisioneel. Hierdie verskil kan toegeskryf word aan die aard van *Delphi* (sien 2.5.2) en die spanning waaronder onderwysers was om die sillabus af te handel.

Onderwysers het wel voorstelle gemaak om die probleme met die oorgang na *Delphi* die hoof te bied. Hierdie voorstelle word vervolgens bespreek.

5.9 ONDERWYSERS SE VOORSTELLE OM DIE OORGANG NA *DELPHI* TE ONDERSTEUN

'n Mens sou verwag dat onderwysers ná 'n jaar van *Scratch*-onderrig daaroor sou reflekteer en voorstelle sou kon gee om die oorgang na *Delphi* te ondersteun. Die meeste aanbevelings het nie verband gehou met hoe *Scratch*-onderrig kan word nie, maar slegs met verskuiwing wat betref tyd, naamlik om vroeër met *Delphi*-onderrig te begin. Onderwysers het gevolglik voorgestel om *Scratch*-onderrig af te water en reeds in graad 10 met *Delphi*-onderrig te begin (sien 4.5).

Daar was wel enkele onderwysers wat ervaar het dat oordrag van programmeringsbeginsels en -begrippe van *Scratch* na *Delphi* plaasgevind het, en hulle het voorgestel dat *Scratch*-onderrig meer pertinent op programmeringsbegrippe moet fokus (sien 4.5).

5.10 AANBEVELINGS MET BETREKKING TOT NAVORSINGSDOELWIT 7: RIGLYNE VIR LEERDERONDERSTEUNING TYDENS DIE ONDERRIG VAN *SCRATCH* OM DIE OORGANG NA *DELPHI* AS PROGRAMMEERTAAL MAKLIKER TE MAAK

Na aanleiding van die voorafgaande bevindings en gevolgtrekkings kan navorsingsdoelwit 7 beantwoord word, naamlik deur riglyne aan onderwysers te gee om leerders tydens die onderrig van *Scratch* te ondersteun ten einde die oorgang na Delphi as programmeertaal vir leerders makliker te maak. Aangesien die proses van programmering, soos in figuur 2.9 uiteengesit, alle aspekte van programmering behels, word aanbevelings eerstens ten opsigte hiervan vir die onderrig van *Scratch* gemaak, en tweedens ten opsigte van die onderrig van spesifieke programmeringsbeginsels en -begrippe. Laastens word aanbevelings ten opsigte van ’n onderrig–leerstrategie vir *Scratch* en die opleiding van onderwysers gemaak. ’n Sintese van alle gevolgtrekkings en aanbevelings word in tabelle 5.2 en 5.3 uiteengesit.

5.10.1 Aanbevelings ten opsigte van die proses van programmering

Die proses van programmering (sien figuur 2.9) behels vier stappe, naamlik probleemvoorstelling, ontwerp, implementering en instandhouding. Drie van die vier stappe, naamlik probleemvoorstelling, ontwerp en instandhouding, kan reeds met *Scratch*-onderrig aangeleer word. Die onderstaande aanbevelings word in dié verband gemaak.

5.10.1.1 Probleemvoorstelling en ontwerp

Die eerste twee stappe van die proses van programmering, naamlik om te bepaal wat ’n program moet doen (stap 1) en ’n oplossing daarvoor deur middel van ’n algoritme voor te stel (stap 2) is universeel vir alle programmeertale (sien 2.7). Algoritme-ontwerp word egter as die moeilikste en belangrikste stap in die proses van programmering beskou (Gomes & Mendes, 2007:1) (sien 2.7). Indien algoritme-ontwerp reeds in *Scratch* aan leerders onderrig word (sien tabel 5.1 en tabel 5.2),

word die moeilikste stap in die proses van programmering reeds in graad 10 onderrig. In graad 11 kan dan slegs hierop voortgebou word.

Daar word aanbeveel dat algoritme-ontwerp in elke nuwe program wat met *Scratch* ontwikkel word, toegepas word (Wang & Zhou, 2011:488) en dat dit nie slegs as ŉ afsonderlike, onafhanklike tema eenmalig gedurende die jaar aangespreek word nie. Wanneer probleemoplossing en algoritme-ontwerp van die begin af, saam met die eenvoudigste probleme geïntegreer word en stelselmatig in moeilikheidsgraad toeneem (Gomes & Mendes, 2007:2,3), sal leerders die tyd en geleentheid kry om probleemoplossings-vaardighede in te oefen (Rogalski & Samurcay, 2010:12).

5.10.1.2 Instandhouding

Die vierde stap in die proses van programmering behels die opspoor van foute en kritiese denke en refleksie oor oplossings (sien figuur 2.9). Leerders moet nie onder ŉ wanindruk gebring word dat *Scratch*-programmering nie foutopsporing en fouthantering insluit nie (sien 2.4.5.3). Deelnemende onderwysers het juis hierdie aspek van die proses van programmering verwaarloos en as gevolg hiervan was foutopsporing en fouthantering een van leerders se grootste struikelblokke in graad 11 (sien 4.3.1.2). Daar word dus aanbeveel dat fouthantering en toetsing reeds tydens die onderrig van *Scratch* behandel moet word en wel op die volgende wyse:

- Algoritmes kan aan toetsing onderwerp word in plaas daarvan om dit eers te implementeer en dan na gelang van programme se afvoer, die sukses daarvan te bepaal.

- Die effektiwiteit van verskillende algoritmes wat dieselfde probleem oplos behoort deur middel van naspeurtabelle (sien 2.5.5.3) getoets te word en besprekings moet daaroor gehou word.

- Leerders moet daarvan bewus gemaak word dat hulle nie tevrede moet wees as programme op die oog af werk nie. Hulle moet geleer en gedurig daaraan herinner word om oor hulle oplossings te reflekteer (sien 2.3), oplossings met ekstreme data te toets (sien 2.5.5.3) en die effektiwiteit van oplossings te bepaal (sien 2.5.4.4), sodat hulle refleksie oor oplossings later as ŉ natuurlike stap

uitvoer (Breed, 2010:243) en hierdie vaardighede ook uiteindelik na *Delphi*-programmering oorgedra word (sien 5.7).

- Leerders moet gedurig blootstelling aan fouthantering kry. Geleenthede moet geskep word om *Scratch*-programme te skep of te bestudeer wat verkeerd werk, of afvoerfoute gee, soos byvoorbeeld wanneer met nul gedeel word (sien 2.4.5.3) of die vierkantswortel van ’n negatiewe getal bepaal word.

Indien bostaande stappe in die proses van programmering reeds in graad 10 onderrig word, behoort dit tot ’n besparing in onderrigtyd in graad 11 te lei (sien 4.2.5 en 4.3.1.3). Al wat dan nog in die proses van programmering onderrig moet word, is die derde stap, naamlik die implementering van oplossings. Hierdie stap is egter programmeertaal-spesifiek (sien 2.3) en dus uniek ten opsigte van die *Scratch*- en *Delphi*-programmeringsomgewings. Die aanbevelings ten opsigte van die implementering van programme fokus gevolglik op spesifieke aspekte rondom programmeertaal en programmeringsbegrippe wat met *Scratch* onderrig kan word.

5.10.2 Aanbevelings ten opsigte van die onderrig van programmerings-begrippe in *Scratch*

Verskeie terminologieë wat algemeen in ’n SOOP soos *Delphi* gebruik word, kan reeds in *Scratch* aan leerders bekend gestel word (sien tabel 2.4). Die doel is juis dat programmering onderrig moet word, en daarom moet die taal van programmering van die begin van graad 10 af gepraat en aan leerders bekend gestel word.

Spesifieke aanbevelings met betrekking tot die onderrig van programmerings-begrippe word vervolgens genoem. Met hierdie aanbevelings word gepoog om programmeringsbegrippe eie aan *Delphi* reeds as deel van *Scratch*-onderrig te behandel sonder dat vooruitverwysings na *Delphi* gemaak word. Sodoende sal leerders in *Delphi* op programmeringsbegrippe wat in *Scratch* aangeleer is, kan voortbou (sien 5.7) en waardevolle onderrigtyd bespaar word.

Aanbevelings met betrekking tot die onderrig van veranderlikes, herhaling- en besluitnemingstrukture, objekgeoriënteerde begrippe en ander begrippe in *Scratch* word vervolgens bespreek.

5.10.2.1 Aanbevelings ten opsigte van die onderrig van veranderlikes

Veranderlikes word konkreet in *Scratch* voorgestel en kan met groot sukses by leerders vasgelê word (sien 2.4.4.4). Verskeie begrippe met betrekking tot veranderlikes, naamlik reikwydte, benoemingskonvensies, datatipes en inisialisering kan reeds met behulp van *Scratch* onderrig word. Die onderstaande aanbevelings word ten opsigte hiervan gemaak:

- Die begrip van reikwydte kan prakties en visueel aan leerders verduidelik word. Die begrippe *lokale* en *globale reikwydte* kan met die opsies van *For this Sprite only* en *For all Sprites* geassosieer word. Die verstekopsie van globale reikwydte (*For all Sprites*) moet nie gedurig gekies word as maklike uitweg nie (sien 2.4.4.4), maar die fokus moet wees op die effektiefste keuse van reikwydte. Daar word dus aanbeveel dat leerders telkens as ŉ veranderlike geskep word, sal besin oor die kwessie van lokale of globale reikwydte en dit deurgaans toepas. Hierdie terminologie moet ook konsekwent gebruik word om graad 10-leerders daarmee vertroud te maak. Dit kan slegs plaasvind indien leerders blootstelling kry aan toepaslike voorbeelde wat hierdie begrippe demonstreer.

- Benoemingskonvensies van veranderlikes, soos dit in *Delphi* gebruik word, kan reeds in *Scratch* geïmplementeer word (sien 2.5.4.3). ŉ Heelgetaltipe (*Integer*) kan aangedui word, deur ŉ **i** vooraan ŉ veranderlike se naam te plaas, en net so kan ŉ tekstipe aangedui word, deur die veranderlike se naam met ŉ **s** te laat begin. Verder moet leerders ook deurgaans bewus gemaak word van die verskil tussen getal- en tekstipes. Ander benoemingskonvensies wat in *Delphi* vereis word, soos byvoorbeeld om nie ŉ spasie in ŉ veranderlike se naam te gebruik nie en om slegs alfabetiese karakters en die "_"-karakter te gebruik, kan ook reeds in *Scratch* toegepas word. Indien leerders van die begin af goeie gewoontes aanleer en bogenoemde benoemingskonvensies gebruik, kan in *Delphi* daarop voortgebou word en hoef addisionele tyd nie in *Delphi*-onderrig bestee te word om hierdie konvensies aan te leer nie.

- Alhoewel datatipes nie spesifiek in *Scratch* gebruik word nie (sien 2.4.4.4), is dit wel moontlik om leerders reeds in *Scratch* aan bepaalde terminologieë met betrekking tot datatipes voor te stel, soos byvoorbeeld *Integer*, *Real*, Boole en

String. Leerders se aandag kan deurgaans op die verskille tussen heelgetal- en reële waardes, karakters en stringe, gevestig word. Leerders se aandag moet ook op die verskillende vorms van blokke en die soorte waardes wat daarin geplaas kan word, gevestig word. (sien tabel 2.3). Weer eens kan die terminologie heeltyd geïnkorporeer word, deur te verwys na ’n Boole-waarde, heelgetal (*Integer*) waarde of tekswaarde, soos van toepassing op die vorms van blokke.

- Die term *inisialisering* moet gedurende *Scratch*-onderrig gebruik en verduidelik word, sodat leerders presies weet wat daarmee bedoel word en nie in ’n situasie beland waar hulle ’n begrip kan toepas, maar nie die terminologie wat daarmee verband hou, ken nie (Meerbaum-Salant *et al*., 2013:244).

Uit bostaande behoort dit duidelik te wees, dat die ooreenkomste tussen die gebruik van veranderlikes in *Scratch* en *Delphi* eintlik merkwaardig is en dat onderwysers tydsgewys nie kan bekostig om die onderrig van veranderlikes na te laat nie. Indien bostaande aanbevelings toegepas word, sal dit leerders ondersteun in hulle oorgang na *Delphi*. Vervolgens word aanbevelings ten opsigte van die onderrig van herhaling- en besluitnemingstrukture in *Scratch* gemaak.

5.10.2.2 Aanbevelings ten opsigte van die onderrig van herhaling- en besluitnemingstrukture

Aanbevelings ten opsigte van die onderrig van herhaling- en besluitnemingstrukture, konsentreer veral op leerders se begrip daarvan – hoe elke struktuur werk, wat die funksie daarvan is en die onderskeid tussen die verskillende strukture. Onderrig moet spesifiek hierop fokus, sodat leerders effektiewe keuses daarvan in die ontwerp van oplossings kan maak. Hulpmiddels soos vloeidiagramme en naspeurtabelle kan met groot vrug gebruik word om die interne werking van die strukture, wat deur *Scratch*-blokke verdoesel word, bloot te lê. Sou die strukture dan later in *Delphi* onderrig word, kan dieselfde vloeidiagramme gebruik word, en is die nodige boustene verskaf sodat slegs ’n nuwe sintaksis aangeleer word. Met betrekking tot herhaling- en besluitnemingstrukture word die volgende spesifieke aanbevelings gemaak:

- Leerders moet op die funksie van die arms van blokke gewys word, naamlik dat dit instruksies saamgroepeer en dus die begin en einde van die strekking van strukture aandui. In *Delphi*-onderrig moet dan terugverwys word na die strukture,

sodat leerders die sintaksis van BEGIN en END kan assosieer met die arms wat die strukture omsluit. Hierdie probleem kan ook die hoof gebied word indien leerders tydens die ontwikkeling van algoritmes en die skryf van pseudokode geleer word om die woorde BEGIN en EINDE sowel as inkeping te gebruik om die strekking van strukture aan te dui.

- Herhaling- en besluitnemingstrukture moet deur middel van vloeidiagramme voorgestel word en naspeurtabelle kan gebruik word om die interne werking van herhalingstrukture en die deursigtigheid van lusbeheerveranderlikes bloot te lê. Die onderrig van herhaling- en besluitnemingstrukture moet hand-aan-hand met vloeidiagramme en naspeurtabelle geskied, om aan leerders die geleentheid te bied om daaroor te reflekteer en die effektiwiteit daarvan te ondersoek.

- Die terminologieë ten opsigte van oneindige, voorwaardelike en onvoorwaardelike herhaling, sowel as inisialiseer, toets en verander, moet in die spreektaal gebruik word wanneer herhalingstrukture onderrig word.

- Die gebruik van die FOREVER-herhalingstruktuur dra min by tot die onderrig van goeie programmeringstegnieke, aangesien dit onvoorwaardelik en oneindig herhaal. Daar word voorgestel dat die FOREVER IF-, REPEAT- en REPEAT UNTIL-herhalingstrukture eerder gebruik word.

- Leerders moet onderrig word om onderskeid tussen die onderskeie herhalingstrukture te kan tref ten einde die effektiefste strukture vir oplossings te kan kies.

- Gebruik die term *voorwaardelike herhalingstrukture* wanneer die REPEAT UNTIL-herhalingstruktuur, of FOREVER IF-struktuur, gebruik word.

- Wys leerders op die beginsels van inisialisering, toetsing en verandering wat in herhalingstrukture voorkom.

- Leerders behoort aangemoedig te word om deur middel van eksperimentering *Scratch* se fasiliteit om uitvoering van programme stap-vir-stap aan te dui, te gebruik.

- Leerders behoort aangemoedig te word om die verskille en ooreenkomste tussen die onderskeie herhalingstrukture te ondersoek.

- Onderrig leerders pertinent in die opstelling van Boole-voorwaardes en die voorkeurorde van AND-, OR- en NOT-operators.

- Gebruik die begrippe WAAR en VALS in die spreektaal en wys leerders op die vorms van die Boole-blokke.

- Aangesien die opstel van Boole-voorwaardes ook in *Scratch* kompleks kan raak wanneer verskeie lae blokke opmekaar gepak word, word aanbeveel dat leerders die Boole-voorwaardes uitskryf en reeds so leer om hakies te gebruik om voorkeurorde aan te dui.

- Onderrig die effektiwiteit van besluitnemingstrukture pertinent aan leerders. Die effektiwiteit van verskeie IF-strukture teenoor die gebruik van geneste IF-ELSE-strukture kan aan leerders uitgewys word deur middel van die stadige uitvoering van *Scratch*-programme.

- Gee aan leerders bestaande oplossings van programme en vra dat hulle die effektiwiteit daarvan moet verbeter, deur die herhaling- of besluitnemingstrukture met meer effektiewe strukture te vervang.

Vervolgens word aanbevelings ten opsigte van die onderrig van objekgeoriënteerde begrippe in *Scratch* gemaak.

5.10.2.3 Aanbevelings ten opsigte van objekgeoriënteerde begrippe

Indien die terme *objek*, *eienskappe*, *gedrag* en *enkapsulasie* nie reeds in *Scratch* by leerders tuisgebring word nie, gaan ’n waardevolle geleentheid verby om objekgeoriënteerde begrippe te onderrig. Op hierdie manier word die basiese begrip van objekte, wat gewoonlik abstrakte en moeilike begrippe vir leerders is (sien 2.7.2), visueel en konkreet uitgebeeld. Daar is reeds aangedui dat leerders wat in *Scratch* aan hierdie begrippe blootgestel word, dit in *Delphi* gouer begryp het (4.3.2.1).

Die volgende aanbevelings word met betrekking tot die onderrig van objekgeoriënteerde begrippe in *Scratch* gemaak:

- ’n Objekgeoriënteerde denkwyse moet so vroeg moontlik gekweek word, aangesien navorsing getoon het dat leerders sukkel om die oorgang na ’n objekgeoriënteerde benadering te maak (Börstler *et al*., 2008:82). Dit word ook duidelik in die KABV gestel dat programmeringsbeginsels reeds in graad 10 binne die objekgeoriënteerde benadering onderrig moet word (DBE, 2011:13). Volgens die KABV moet die “programmeringsomgewing deur die gebruik van geanimeerde karakters as objekte verken [word], d.i. hul status en gedrag” (DBE, 2011:21).

- Al is *Scratch* nie ’n ware objekgeoriënteerde programmeertaal nie, is daar bepaalde begrippe rakende objekte wat reeds gevestig kan word. *Sprites* kan objekte genoem word. Hulle het eienskappe, soos byvoorbeeld X-koördinate en Y-koördinate, en ook gedrag, aangesien hulle bepaalde aksies kan uitvoer.

- Die beginsels van enkapsulasie by objekte kan in *Scratch* reeds in ’n beperkte mate oorgedra word deur die gebruik van *For this Sprite only*. Instruksies kan slegs in die *Sprites* waarin dit voorkom uitvoer, en *Sprites* het nie toegang tot veranderlikes in ander *Scripts* nie, tensy dit op die *Stage* geplaas word (Maloney *et al*., 2010:10).

- Begrippe rakende objekte kan uitgebrei word na objekte in die omliggende wêreld vir verdere versterking daarvan (sien 2.7.2).

Behalwe dat al bostaande aanbevelings geïmplementeer word, moet dit steeds ondersteun word deur ’n toepaslike onderrig–leerstrategie. Vervolgens word aanbevelings ten opsigte van ’n onderrig–leerstrategie bespreek.

5.10.3 Aanbevelings ten opsigte van ’n onderrig–leerstrategie vir *Scratch*

Alhoewel sekere programmeringsbegrippe in *Scratch* deur leerders self ontdek kan word, moet onderrig nie aan selfontdekking oorgelaat word nie, aangesien navorsing bewys het dat swak programmeringsgewoontes so kan ontstaan en ’n diepere begrip van programmeringsbeginsels en -begrippe dan ontbreek (Meerbaum-Salant *et al*., 2011) (sien 2.8.1). Die aanbeveling word dus gemaak ten opsigte van ’n onderrig–leerstrategie waar leerders hulle kreatiewe denke en probleemoplossings-vaardighede kan ontwikkel, en leer om saam met ander leerders te werk (MIT Media

Lab, 2003). Paarprogrammering (sien 2.7.1.2) (Mentz *et al.,* 2008:259–260), koöperatiewe leer (sien 2.7.1.1) en probleem-gebaseerde leer (sien 2.7.1.3) is suksesvolle onderrig–leerstrategieë wat aan bogenoemde vereistes voldoen

Al bostaande aanbevelings sal egter van geen waarde wees indien die opleiding van onderwysers nie aandag geniet nie.

5.10.4 Aanbevelings ten opsigte van die opleiding van onderwysers

Die verantwoordelikheid van ’n IT onderwyser is groot – ’n moeilike, vol sillabus, en ’n veld wat gereeld verander. Graad 10, 11 en 12 word verder in die meeste skole deur slegs een IT-onderwyser onderrig. IT-onderwysers moet ook as gevolg van lae leerdergetalle ander vakke onderrig en het ’n vol buitemuurse program, wat hulle werkslading nog groter maak. Hulle het nie tyd om ander onderrig–leerstrategieë na te vors nie en kan nie bekostig om daarmee te eksperimenteer nie. Geleenthede moet dus geskep word om hierdie onderwysers te ondersteun sodat hulle op ’n gereelde basis idees kan uitruil en oor hulle onderrig kan reflekteer. Dit is juis op dié gebied dat hierdie studie ’n waardevolle bydrae ten opsigte van die onderrig van *Scratch* kan lewer.

Nuwe programmeertale soos *Scratch* vereis nuwe onderrig–leerstrategieë en spesifieke onderrigbenaderings vir suksesvolle onderrig daarvan (sien 2.8.1). Onderwysers benodig egter ook voldoende opleiding hierin. Kölling (2008:99) en Meerbaum-Salant *et al*. (2013:171) voer aan dat, indien onderwysers nie behoorlik opgelei word om nuwe programmeertale te onderrig nie, hulle nie weet hoe om leerders te onderrig nie en gevolglik ontstaan swak programmeringsgewoontes. Dieselfde bevinding is tydens hierdie navorsing gemaak. Selfs ervare IT-onderwysers was onseker hoe om *Scratch* te onderrig en het ’n behoefte aan alternatiewe onderrig–leerstrategieë getoon. Daar word dus aanbeveel dat die Departement van Basiese Onderwys meer geleenthede skep om onderwysers op te lei ten opsigte van die onderrig van *Scratch* en onderrig–leerstrategieë soos paarprogrammering, koöperatiewe leer en probleemgebaseerde leer met die oog op die bemeestering van programmeringsbeginsels en –begrippe.

5.10.5 Opsomming van aanbevelings van die navorsing

In die literatuur is bevind dat studies gedoen is wat aangetoon het dat programmeringsbeginsels en –begrippe wel deur *Scratch* aangeleer kan word, maar dat *Scratch* op ’n bepaalde manier onderrig moet word vir maksimale begrip van programmeringsbeginsels en -begrippe. In die empiriese ondersoek is bevind dat onderwysers nie bepaalde programmeringsbeginsels en –begrippe onderrig het nie, maar dat hulle dit wel in die toekoms so gaan doen. ’n Opsomming van alle aanbevelings van hierdie studie, wat gebaseer is op die literatuurstudie en resultate voortspruitend uit die empiriese navorsing, word vervolgens in tabel 5.2 uiteengesit

Tabel 5.2: Opsomming van aanbevelings ten opsigte van die onderrig van *Scratch*

1. Aanbevelings vir die onderrig van die proses van programmering	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Probleemvoorstelling en algoritme-ontwerp	• Integreer deurgaans algoritme-ontwerp met elke nuwe program wat geskryf word. • Gebruik alledaagse, opwindende voorbeelde wat op leerders se leefwêreld van toepassing is, soos byvoorbeeld hoe om in rugby te skrum, leerders se optrede wanneer die klok aan die einde van die periode lui, omruil van 'n pap wiel, aansoek om ID-dokument. • Laat leerders self probleemscenario's uitdink waarin bepaalde programmeringsbegrippe voorkom, en algoritmes daarvoor op stel. • Bestudeer bestaande algoritmes en vra leerders om: o oor die effektiwiteit daarvan te reflekteer; o dit deur middel van naspeurtabelle te toets deur ekstreme data in te voer; o programmeringsbegrippe wat daarin voorkom, uit te wys; o dit na vloeidiagramme om te skakel en omgekeerd; en o daarop uit te brei en nuwe programmeringsbegrippe by te voeg. • Die effektiwiteit van verskillende algoritmes wat dieselfde probleem oplos, behoort deur middel van naspeurtabelle getoets te word en in groepe bespreek te word. • Tydens die skryf van pseudokode moet inkeping en die woorde BEGIN en EINDE gebruik word om die strekking van strukture aan te dui.

1. Aanbevelings vir die onderrig van die proses van programmering	
Programmeringsbeginsel/-begrip	**Aanbeveling**

Instandhouding, foutopsporing en fouthantering	• Foutopsporing en fouthantering moet deurgaans met die skryf van oplossings geïntegreer word.
	• Leerders moet by *Scratch* soveel moontlik blootstelling aan foutopsporing en fouthantering kry.
	• Skep bewustheid daarvan dat programme wat op die op die oog af werk, nie noodwendig probleme korrek oplos nie:
	  ○ toets afvoer met ’n verskeidenheid toetsdata, en vergelyk dit met handberekenings; en
	  ○ toets altyd met ekstreme data.
	• Moedig leerders aan om toevoerdata te valideer.
	• Gee blootstelling aan programme wat foutief uitvoer, soos byvoorbeeld waar met nul gedeel word of die vierkantswortel van ’n negatiewe getal bereken word. Leerders moet self programme probeer uitvoer en bepaal wat die fout is.
	• Skep geleenthede om oor oplossings te reflekteer:
	  ○ Leerders kan mekaar se programme evalueer en met ekstreme en ongeldige data toets.
	  ○ Bepaal effektiwiteit van oplossings deur uitvoering daarvan met naspeurtabelle na te gaan, of deur effektiwiteit met behulp van uitvoer in stadige aksie te evalueer – leerders kan mekaar se programme so evalueer en voorstelle gee om daarop te verbeter.

2. Aanbevelings vir die onderrig van programmeringsbeginsels en -begrippe	
2.1 Veranderlikes	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Benoemingskonvensies van veranderlikes	• Implementeer benoemingskonvensies soos dit in *Delphi* gebruik word. 'n Heelgetaltipe (*Integer*) kan aangedui word, deur 'n **i** vooraan 'n veranderlike se naam te plaas, en net so kan 'n tekstipe (String) aangedui word deur die veranderlike se naam met 'n **s** te laat begin. • Gebruik slegs toelaatbare benoemingskonvensies betreffende die gebruik van karakters in veranderlike name, soos wat in *Delphi* die gebruik is. Gebruik byvoorbeeld die "_"-karakter in plaas van spasies in veranderlikes se name en moenie veranderlikes se name met syfers laat begin nie. • Leerders moet aangemoedig word om beskrywende name vir veranderlikes te gebruik. Name moet betekenisvol wees ten opsigte van die doel van die veranderlike.
Reikwydte van veranderlikes	• Gebruik die terminologie *lokale* en *globale reikwydte* wanneer hierdie begrippe verduidelik word. • Assosieer die begrippe *lokale* en *globale reikwydte* met die gebruik van *For this Sprite only* en *For all Sprites*. • Die verstekopsie van globale reikwydte (*For all Sprites*) moet nie telkens gekies word nie, maar die fokus moet altyd op die effektiefste keuse van reikwydte wees. • Gee blootstelling aan toepaslike voorbeelde wat reikwydte demonstreer.

2.1 Veranderlikes ..... (vervolg)	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Datatipes	• Gebruik, waar van toepassing, soveel moontlik terminologieë wat met datatipes verband hou, naamlik heelgetal (*Integer*), reëel (*Real*), tekstipe (*String*), karakter en Boole. • Wys leerders op die verskil tussen heelgetalle (*Integer*) en reële getalle (*Real*). • Vestig leerders se aandag op die verskillende vorms van parameter-gleuwe en die soorte waardes wat daarin geplaas kan word. Inkorporeer terminologieë deur na ’n Boole-tipe waarde, String-tipe en getal-waarde soos van toepassing op die vorms van blokke te verwys: ○ *Boole-waardes* word deur blokke met hoekige kante voorgestel ○ *Numeriese waardes* word deur blokke met ronde kante voorgestel ○ *Tekswaardes* word deur reghoekige blokke voorgestel
Inisialisering van veranderlikes	• Gebruik die term *inisialisering* wanneer spesifieke waardes aanvanklik aan veranderlikes toegeken word, sodat leerders weet wat die term beteken. • Vestig leerders se aandag daarop dat die waarde van ’n veranderlike in *Scratch* nie noodwendig altyd nul is nie, maar dat dit die vorige waarde behou en dat inisialisering nodig is. • Wys inisialisering uit deur na die inhoud van veranderlikes in monitors te verwys. In die onderstaande voorbeeld, word die veranderlike X byvoorbeeld na nul geïnisialiseer: 

2.2 Herhalingstrukture	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Begin en einde van strukture	• Wys leerders telkens op die funksie van die blokke se arms – dat dit die instruksies saamgroepeer en dus die begin en einde van strukture aandui. • Vra leerders om die pseudokode van bestaande *Scratch*-oplossings neer te skryf, en lê klem op die gebruik van inkeping en die woorde BEGIN en EINDE om die strekking van strukture aan te dui.
Interne werking van herhaling	• Stel herhalingstrukture deur middel van vloeidiagramme voor en gebruik ook naspeurtabelle om die interne werking van herhalingstrukture en die deursigtigheid van lusbeheerveranderlikes bloot te lê. In die onderstaande kodeblok, wat instruksies tien keer gaan herhaal, is daar 'n teller wat geïnisialiseer word, telkens inkrementeer en teenoor die waarde 10 getoets word. Hierdie interne werking van die lus moet aan leerders verduidelik word.  • Leerders moet dus reeds in *Scratch* vloeidiagramme kan teken wat die interne werking van lusbeheerveranderlikes en toetsing voorstel.

2.2 Herhalingstrukture ..... (vervolg)	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Oneindige, voorwaardelike en onvoorwaardelike herhaling	• Vermy die gebruik van FOREVER. Gebruik eerder FOREVER-IF, REPEAT en REPEAT-UNTIL. • Die verskil tussen die werking van bogenoemde lusstrukture moet goed aan leerders verduidelik word. Die stadige uitvoerfunksie van *Scratch*, tesame met monitors, kan gebruik word om die werking van herhaling aan te dui. • Stel leerders bekend aan terminologieë, soos *onvoorwaardelike* (REPEAT) en *voorwaardelike* (FOREVER-IF en REPEAT-UNTIL) *herhaling*. Dui die verskil tussen voorwaardelike en onvoorwaardelike herhaling tydens onderrig aan en lei leerders om dit in hulle oplossings korrek toe te pas. • Reflekteer oor oplossings wat nie effektiewe herhalingstrukture gebruik nie. • Gee aan leerders voorbeelde waarin oneindige herhaling voorkom en laat hulle dit evalueer en die nodige foutopsporing en fouthantering doen. • Gee twee verskillende programme wat dieselfde probleem oplos, aan leerders. Laat leerders in groepe, deur middel van *Scratch* se stadige-uitvoer-opsie, die uitvoering ten opsigte van verskille, ooreenkomste en effektiwiteit evalueer. • Stel twee verskillende herhalingstrukture deur middel van vloeidiagramme voor om die verskille tussen die werking van die strukture uit te wys.

2.2 Herhalingstrukture ..... (vervolg)	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Oneindige, voorwaardelike en onvoorwaardelike herhaling (vervolg)	• Wys leerders telkens op die beginsels van inisialisering, toetsing en verandering wat in herhalingstrukture voorkom. Die gebruik van vloeidiagramme en pseudokode kan waardevol wees in dié verband. Gee aan leerders pseudokode waarin van hierdie beginsels weggelaat is en laat hulle die uitwerking daarvan met naspeurtabelle evalueer, sowel as oplossings vir die foutiewe werking voorstel. • Die fokus behoort altyd op die gebruik van die effektiefste herhalingstrukture te wees.
2.3 Besluitneming en Boole-voorwaardes	
Boole-voorwaardes, AND-, OR- en NOT-operators	• Onderrig leerders in die samestelling van Boole-voorwaardes en die voorkeurorde van AND-, OR- en NOT-operators. • Gee voorbeelde uit die spreektaal en vra leerders om Boole-voorwaardes daarvoor op te stel. • Begin by eenvoudige uitdrukkings en volg op met uitdrukkings wat al moeiliker word, om die voorkeurorde van operators duidelik aan te dui. • Gee aan leerders geskrewe voorbeelde van Boole-uitdrukkings, in verskillende moeilikheidsgrade, sodat hulle die uitkomste daarvan moet bepaal.

2.3 Besluitneming en Boole-voorwaardes ..... (vervolg)	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Boole-voorwaardes, AND-, OR- en NOT-operators **(vervolg)**	• Gee voorbeelde waar die voorkeurorde van operators nie met hakies afgedwing word nie, sodat leerders die effek van uitvoering ten opsigte van verstekvoorkeurorde kan sien. • Aangesien die opstel van Boole-voorwaardes kompleks kan raak wanneer verskeie lae blokke opmekaar gepak word, word daar aanbeveel dat leerders die Boole-voorwaardes uitskryf en so leer om hakies te gebruik om voorkeurorde aan te dui. • Gebruik die begrippe WAAR en VALS in die spreektaal en vestig leerders se aandag op die vorms van die Boole-blokke. • Gee aan leerders bestaande programme en vra dat hulle alle voorkomste van Boole-uitdrukkings in die program moet aandui (in herhalingstrukture, in besluitnemingstrukture en ander blokke).

2.3 Besluitneming en Boole-voorwaardes ..... (vervolg)	
Programmeringsbeginsel/-begrip	**Aanbeveling**
IF- en geneste IF-ELSE-strukture	• Fokus op die effektiwiteit van verskillende oplossings vir dieselfde probleme. Die effektiwiteit van verskeie IF-strukture teenoor die gebruik van geneste IF-ELSE-strukture kan deur leerders nagegaan word deur middel van die stadige uitvoering van *Scratch*-programme. • Gee bestaande oplossings aan leerders en versoek dat hulle dit met meer effektiewe oplossings met betrekking tot die besluitnemingstrukture moet vervang. • Vra leerders om die pseudokode van bestaande *Scratch*-oplossings neer te skryf en lê klem op die gebruik van inkeping en die woorde BEGIN en EINDE waar nodig. • Fokus op die gebruik van die arms om die begin en einde van die instruksies wat in die keusestrukture ingesluit word, aan te dui.

2.4 Objekgeoriënteerde begrippe	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Objekte, gedrag, eienskappe	• Verwys na objekte in die wêreld en gee voorbeelde van die eienskappe en gedrag van hierdie objekte. • Vra leerders om self, in groepsverband, voorbeelde van objekte in hulle leefwêreld te gee en die eienskappe en gedrag daarvan neer te skryf, en hou ’n groepbespreking daaroor. • Verwys na *Sprites* as objekte. • Wys leerders daarop dat objekte (*Sprites*) gedrag het, byvoorbeeld vorentoe kan beweeg en kan draai. • Wys leerders daarop dat objekte *(Sprites)* eienskappe het, byvoorbeeld X- en Y-koördinate. • Vra leerders om in groepe voorbeelde van eienskappe en gedrag van *Sprites* neer te skryf en hou ’n klasbespreking daaroor. • Hierdie terminologieë moet nie slegs eenmalig gebruik word nie, maar deurgaans, waar moontlik. In plaas daarvan dat onderwysers byvoorbeeld van ’n *Sprite* praat, kan die woord *objek* eerder gebruik word.
Enkapsulasie	• ’n Objek (*Sprite*) kan veranderlikes versteek (enkapsuleer) sodat ander objekte (*Sprites*) dit nie kan raaksien nie, deur ’n veranderlike te verklaar met die *For this Sprite only*-opsie. • Stel die term enkapsulasie aan leerders bekend en reflekteer daaroor deur byvoorbeeld te vra hoekom dit nodig sou wees.

2.4 Objekgeoriënteerde begrippe ..... (vervolg)	
Programmeringsbeginsel/-begrip	**Aanbeveling**
Funksies en parameteroordrag	• Daar bestaan verskeie funksies in *Scratch*, soos byvoorbeeld PICK RANDOM, JOIN, LENGTH OF, ROUND en SQRT. Wanneer hierdie funksies onderrig word, moet toepaslike terminologie gebruik word – noem dit *funksies* en wys leerders op die gebruik van *argumente* en *parameters*. • Verduidelik aan leerders: ○ wat ’n funksie is; ○ dat ’n funksie deur middel van ’n roepinstruksie geroep word; ○ dat ’n funksie ’n bepaalde antwoord verskaf na gelang van die waarde van die parameter wat daaraan gegee word; ○ dat ’n funksie ’n spesifieke soort antwoord gee (getal, teks of Boole); en ○ dat funksies nie as aparte instruksies gebruik kan word nie, maar binne-in ander instruksieblokke geplaas moet word. • Wys leerders daarop dat funksies verskillende aantal parameters kan ontvang (vergelyk JOIN en ROUND), maar slegs een waarde as antwoord kan gee. • Let op die vorms van die funksieblokke en koppel dit aan datatipes. • Laat leerders funksies in *Scratch* of in ’n bestaande *Scratch*-program identifiseer, neerskryf watter parameters dit ontvang en kommentaar lewer oor die tipes van die parameters, sowel as die tipe waarde wat die funksie terugstuur. • Vra leerders om te evalueer of BROADCAST ’n funksie is: (BROADCAST kan nie as ’n funksie beskou word nie, omdat dit as ’n onafhanklike instruksie gebruik kan word en nie ’n antwoord op ’n bewerking bereken nie. Dit kan wel as ’n metode beskou word om ’n boodskap [parameter] na ander *Sprites* oor te dra.)

2.5 Bewerkingsoperators	
Programmeringsbeginsel/-begrip	**Aanbeveling**
+, - , * / MOD, hakies	• Gee verskeie voorbeelde van programme wat die gebruik en voorkeurorde van alle bewerkingsoperators bevat, in verskillende moeilikheidsgrade. • Gee aan leerders 'n toepassing om te bepaal hoeveel keer 'n getal in 'n ander kan indeel (om voor te berei op die DIV-operator van *Delphi*) en wat die res is wanneer die deling plaasgevind het. • Gee aan leerders toepassings wat met hul ervaringswêreld verband hou en wat berekenings bevat, sowel as berekenings wat in ander vakke soos wiskunde, rekeningkunde, geografie en fisiese wetenskap voorkom. • Laat leerders self aan toepassings vir bogenoemde dink.
2.6 Karakterhantering	
Karakterhanteringsfunksies en bewerkings met stringe	• Gebruik die terme *karakter* en *string* en vestig die begrippe. • Verskeie aspekte van karakterhantering kan reeds met *Scratch* onderrig word deur die gebruik van JOIN, LETTER OF en LENGTH OF.

3. ’n Onderrig–leerstrategie vir *Scratch*
Volg ’n aktiewe, leerdergesentreerde onderrig–leerstrategie, waaronder die volgende: • Paarprogrammering gekombineer met die vyf beginsels van koöperatiewe leer, naamlik positiewe interafhanklikheid, individuele verantwoordelikheid, bevorderende persoonlike interaksie, sosiale vaardighede en groepnadenke. • Probleemgebaseerde leer. • Ontdekkende leer.

Met bostaande aanbevelings in ag geneem, kan die navorser nie met onderwysers se pragmatiese voorstel om die *Scratch*-onderrigjaar in te kort ten einde vroeër met *Delphi*-onderrig te kan begin, saamstem nie. *Scratch*-onderrig voorsien dus die nodige kapasiteit om programmeringsbeginsels en -begrippe te vestig, sonder die kompleksiteit van *Delphi*. *Scratch*-onderrig met die oog op *Delphi* verg deeglike beplanning en voorbereiding van die onderwyser ten einde al bostaande aanbevelings in onderrig te inkorporeer. Dit sou egter onregverdig wees om te verwag dat onderwysers bostaande aanbevelings sonder behoorlike opleiding en geleenthede vir refleksie in hulle onderrig sou kan toepas.

Ander sake wat nie direk op die navorsingsdoelwitte gerig is nie, maar ook met die onderrig van *Scratch* verband hou, het in die navorsing na vore gekom, naamlik die gebruik van handboeke en die opleiding van onderwysers. Tabel 5.3 bied 'n opsomming van die aanbevelings in dié verband.

Tabel 5.3: Addisionele aanbevelings voortspruitend uit die navorsing

Aanbevelings ten opsigte van die gebruik van handboeke en die KABV
• Handboeke behoort nie slaafs gebruik te word en van voor na agter deurgewerk te word nie. Indien daar in handboeke, byvoorbeeld, 'n hele hoofstuk oor algoritme-ontwerp of foutopsporing en -hantering is, moet hierdie hoofstukke nie op 'n spesifieke tyd van die jaar afsonderlik behandel word nie, maar dié twee onderafdelings moet van die begin af by die ontwikkeling van alle oplossings geïntegreer word.
• Onderwysers moet hulle deeglik vergewis van programmeringsbeginsels en -begrippe wat volgens die KABV onderrig moet word. Verskeie hulpmiddels, en nie net 'n spesifieke handboek nie, moet gebruik word om sodanige begrippe te onderrig.
• Indien voorgeskrewe werk nie in handboeke voorkom nie, moet dit nie uitgelaat word nie, maar addisionele bronne moet geraadpleeg word.

Aanbevelings ten opsigte van die gebruik van handboeke en die KABV ..... (vervolg)
• Maak gebruik van gebeure uit die werklike lewe en uit leerders se ervaringswêreld om programmeringsprobleme op te stel. Voorbeelde hoef nie altyd handboekgebonde te wees nie. Leerders kan gevra word om self ook programmeringsprobleme waarin bepaalde begrippe voorkom, uit te dink soos byvoorbeeld om programme te skryf wat hulle skool bemark, tellings hou tydens rugby- of krieketwedstryde, of stemme tel tydens 'n verkiesing.
Aanbevelings ten opsigte van die opleiding van onderwysers
• Opleiding van IT-onderwysers moet hoër prioriteit geniet by die Departement van Basiese Onderwys. • Jaarlikse opknappingskursusse ten opsigte van die onderrig van *Scratch* en *Delphi*, sowel as nuwe onderrig–leerstrategieë moet aangebied word. • Geleenthede moet geskep word waartydens onderwysers oor hulle onderrig kan reflekteer en idees met mekaar kan uitruil.

5.11 TEKORTKOMINGE VAN DIE STUDIE

Hierdie kwalitatiewe studie was slegs van toepassing op geselekteerde onderwysers in Noordwesprovinsie en het ook slegs onderwysers uit skole waar *Scratch* in graad 10 en *Delphi* as programmeertaal in graad 11 onderrig word, betrek. Gevolglik kan veralgemenings nie uit hierdie studie gemaak word na alle IT-onderwysers in Noordwesprovinsie nie. Alhoewel dit aanvaar kan word dat ander provinsies soortgelyke probleme ervaar het, kan hierdie aanbevelings ook nie sonder meer op alle IT-onderrig in Suid-Afrika van toepassing gemaak word nie. IT-onderwysers in ander provinsies het ook verskillende opleiding ontvang.

5.12 AANBEVELINGS VIR VERDERE NAVORSING

Die studie kan uitgebrei word na ander provinsies, sowel as provinsies waar *Scratch* in graad 10 en *Java* in graad 11 onderrig word. Die empiriese data is in 2013 ingesamel, ná die eerste jaar van die implementering van die KABV in 2012 vir graad

10-leerders. Dit kan herhaal word ten einde te bepaal wat onderwysers se huidige ervaring is en of hulle hulle onderrig aangepas het in 2014. Aangesien hierdie studie op onderwysers se ervaring van die oorgang na *Delphi* gekonsentreer het, kan navorsing ook gedoen word om te bepaal hoe leerders die onderrig van *Scratch* en die oorgang na *Delphi* ervaar.

5.13 TEN SLOTTE

In hierdie studie is die aard van programmering in *Scratch* en *Delphi* en die onderrig daarvan aan die hand van die literatuur ondersoek. Tydens ŉ empiriese studie is ondersoek ingestel hoe *Scratch* onderrig word en hoe geselekteerde onderwysers die oorgang na *Delphi* ervaar het. Daar is bevind dat *Scratch* nie pertinent onderrig is met die oog op die oorgang na *Delphi* nie. Die gevolg hiervan was dat min programmeringsbeginsels en –begrippe oorgedra is na *Delphi* en dat graad 11 leerders nie gereed was om met gevorderde programmering in *Delphi*, soos deur die KABV voorgeskryf, te begin nie. Onderwysers was gevolglik onder groot druk om weer bepaalde programmeringsbeginsels en –begrippe in graad 11 te onderrig sodat die sillabus betyds afgehandel kon word. Aanbevelings ten opsigte van die onderrig van *Scratch* is na aanleiding van die literatuurstudie en empiriese ondersoek gemaak ten einde aan leerders ondersteuning te bied tydens die oorgang na *Delphi*.

Daar moet in gedagte gehou word dat die empiriese studie direk ná die eerste jaar van *Scratch*-onderrig gedoen is en dat die resultate en aanbevelings in die lig van hierdie tydsbestek gedoen is. Die bevindings van hierdie studie behoort by te dra tot meer effektiewe onderrig van *Scratch* in graad 10 met die oog op die oorgang na *Delphi* in graad 11.

BRONNELYS

ANON. 2010. Data, data everywhere. *The Economist*, 25 Feb. http://www.economist.com/node/15557443 Date of access: 25 May 2013.

ARMONI, M., GAL-EZER, J. & HAZZAN, O. 2006. Reductive thinking in computer science. *Computer science education*, 16(4):281–301.

ASTRACHAN, O., BRUCE, K., KOFFMAN, E., KÖLLING, M. & REGES, S. 2005. Resolved: objects early has failed. *ACM SIGCSE bulletin*, 37(1):451–452.

BABBIE, E. & MOUTON, J. 2001. The practice of social research. Cape Town: Oxford University Press Southern Africa 674 p.

BAGLEY, C.A. & CHOU, C.C. 2007. Collaboration and the importance for novices in learning Java computer programming. *ACM SIGCSE bulletin*, 39(3):211–215.

BALDWIN, L.P. & KULJIS, J. 2000. Visualisation techniques for learning and teaching programming. *Journal of computing and information technology*, 8(4):285–291.

BECK, L.L. & CHIZHIK, A.W. 2008. An experimental study of cooperative learning in CS1. *ACM SIGCSE bulletin*, 40(1):205–209.

BENNEDSEN, J. 2008. Introduction to Part I. (*In* Bennedsen, J., Caspersen, M.E. & Kölling, M., *eds*. Reflections on the teaching of programming. Berlin: Springer-Verlag. p. 3–5.)

BENNEDSEN, J. & CASPERSEN, M.E. 2008. Exposing the programming process. (*In* Bennedsen, J., Caspersen, M.E. & Kölling, M., *eds*. Reflections on the teaching of programming. Berlin: Springer-Verlag. p. 6–16.)

BERGIN, J. 2009. Why procedural is the wrong first paradigm if OOP is the goal. http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html Date of access: 6 Mar. 2013.

BÖRSTLER, J., NORDSTRÖM, M., KALLIN, W.L., MOSTRÖM, J. & ELIASSON, J. 2008. Transition to OOP/Java: a never ending story. (*In* Bennedsen, J., Caspersen, M.E. & Kölling, M., *eds*. Reflections on the teaching of programming. Berlin: Springer-Verlag. p. 80–97.)

BREED, E.A. 2010. 'n Metakognitiewe onderrigleerstrategie vir paarprogrammeerders ter verbetering van kennisproduktiwiteit. Potchefstroom: NWU. (Thesis – PhD.) 320 p.

BRUCE, K.B. 2004. Controversy on how to teach CS1: a discussion on the SIGCSE-members mailing list. *The SIGCSE bulletin*, 36(4):29–34.

BRYANT, S., ROMERO, P. & DU BOULAY, B. 2008. Pair programming and the mysterious role of the navigator. *International journal of human-computer studies*, 66(7):519–529.

CARVER, S.M. 1986. Transfer of LOGO debugging skill: analysis, instruction and assessment. Pittsburgh, PA: Carnegie Mellon University. (Thesis – PhD.) 135 p.

CHI, M.T.H., DE LEEUW, N., CHIU, M.H. & LAVANCHER, C. 1994. Eliciting self-explanations improves understanding. *Cognitive science*, 18(3):439–477.

CLANCY, M. 2004. Misconceptions and attitudes that interfere with learning to program. (*In* Fincher, S. & Petre, M., *eds*. Computer science education research. Lisse: Taylor & Francis. p. 85–100.)

CLEAR, T. 1997. The nature of cognition and action. *ACM SIGCSE bulletin*, 29(4):25–29.

CRESWELL, J.W. 2008. Educational research: planning, conducting and evaluating quantitative and qualitative research. 3rd ed. Upper Saddle River, NJ: Pearson. 670 p.

CRESWELL, J.W. 2009. Research design: qualitative, quantitative and mixed methods approaches. 3rd ed. Thousand Oaks, CA: Sage. 260 p.

DBE (Department of Basic Education). 2011. Kurrikulum- en assesseringsbeleidsverklaring. Pretoria: Government Printers. 60 p.

DEE, J. 2013. My reflections from the April 13, 2013 ScratchEd meetup. http://scratched.media.mit.edu/resources/my-reflections-april-13-2013-scratched-meetup Date of access: 28 Apr. 2013.

DEEK, F.P. 1999. A framework for an automated problem solving and program development environment. *Journal of integrated design and process science*, 3(3):1–13.

DÉTIENNE, F. 2010. Acquire experience in object-oriented programming: effects on design strategies. (*In* Lemut, E., Du Boulay, B. & Dettori, G., *eds*. Cognitive models and intelligent environments for learning programming. Berlin: Springer-Verlag. p. 49–58.)

DIJKSTRA, E.W. 1989. On the cruelty of really teaching computer science (The SIGCSE Award Lecture 1989). *ACM SIGCSE bulletin*, 32:1398–1404.

DoE (Department of Education). 2003. National curriculum statement. Pretoria: Shumani Printers. 66 p.

DONCHEV, I. & TODOROVA, E. 2013. Training in object-oriented programming and C++11. *Computer and information science*, 6(2):84–92.

DU BOULAY, B. 1989. Some difficulties of learning to program. (*In* Du Boulay, B., *ed*. Studying the novice programmer. Hillsdale: Lawrence Erlbaum. p. 283–299.)

DUCH, B.J., GROH, S.E. & ALLEN, D.E. 2001. Why problem-based learning? (*In* Duch, B.J., Groh, S.E. & Allen, D.E., *eds*. The power of problem-based learning: a practical "how to" for teaching undergraduate courses in any discipline. Sterling, VA: Stylus. p. 3–12.)

FINCHER, S., COOPER, S., KÖLLING, M. & MALONEY, J. 2010. Comparing Alice, Greenfoot and Scratch. ACM Transactions on Computing Education (TOCE), 10(4):192–193.

FORD, J.L. 2009. Scratch programming for teens. Toronto: Course Technology. 315 p.

FRIESE, S. 2012. Qualitative data analysis with ATLAS.ti. London: Sage. 274 p.

GAO, R. 2011. Reforming to improve the teaching quality of computer programming language. (*In* Computer Science & Education (ICCSE), 6th International Conference on IEEE, Singapore. IEEE. p. 1267–1269.)

GIBBS, G.R. 2007. Analyzing qualitative data. London: Sage. 160 p.

GIBSON, J.P. & O'KELLY, J. 2005. Software engineering as a model of understanding for learning and problem solving. (*In* Proceedings of the First International Workshop on Computing Education Research, Seattle. NY: ACM. p. 87–97.)

GOMES, A. & MENDES, A. 2007. Learning to program-difficulties and solutions. Paper presented at the International Conference on Engineering Education, Coimbra, 3–7 September. http://ineer.org/Events/ICEE2007/papers/411.pdf Date of access: 28 Aug. 2012.

GRANT, N.S. 2003. A study on critical thinking, cognitive learning style and gender in various information science programming classes. (*In* Proceedings of the 4th Conference on Information Technology Curriculum, West Lafayette. NY: ACM. p. 96–99.)

GUGLIELMINO, L.M. 2003. Contributions of the International Self-Directed Learning Symposia. *Adult learning*, 14(4):23–25.

GUZDIAL, M. 2004. Programming environments for novices. (*In* Fincher, S. & Petre, M., *eds*. Computer science education research. Lisse: Taylor & Francis. p. 128–154.)

HARVEY, B. & MÖNIG, J. 2010. Bringing no ceiling to Scratch: can one language serve kids and computer scientists? Paper presented at Constructionism 2010, Paris, 16 August. http://www.eecs.berkeley.edu/~bh/BYOB.pdf Date of access: 13 Aug. 2014.

HAVENGA, H.M. 2008. An investigation of students' knowledge, skills and strategies during problem solving in object oriented programming. Pretoria: UNISA. (Thesis – PhD.) 306 p.

ISMAIL, M.N., NGAH, N.A. & UMAR I.N. 2010. Instructional strategy in the teaching of computer programming: a need assessment analyses. *The Turkish online journal of educational technology*, 9(2):125–131.

JEHNG, J.C., TUNG, S.H. & CHANG, C.T. 1999. A visualisation approach to learning the concept of recursion. *Journal of computer assisted learning*, 15(4):279–290.

JENKINS, T. 2002. On the difficulty of learning to program. (*In* Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, vol 4, Loughborough. LTSN-ICS. p. 53–58.)

JOHNSON, D.W., JOHNSON, R.T. & HOLUBEC, E. 2008. Cooperation in the classroom. 8th ed. Edina: Interaction Book Company.

KAPSIMALI, V. & SAMPSON, D. 2011. A pilot case study using Scratch in school education. (*In* Sotiriou, S. & Szücs, A., *eds*. Never waste a crisis! 2011 EDEN Open Classroom Conference, Ellinogermaniki Agogi, Athens. Budapest: EDEN. p. 184–187.)

KELLEHER, C. & PAUSCH, R. 2005. Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. *ACM computing surveys*, 37(2):83–137.

KERMAN, M.C. 2002. Programming and problem solving with Delphi. Harlow: Pearson Education.

KÖLLING, M. 2008. Using BlueJ to introduce programming. (*In* Bennedsen, J., Caspersen, M.E. & Kölling, M., *eds*. Reflections on the teaching of programming. Berlin: Springer-Verlag. p. 98–115.)

LAHTINEN, E., ALA-MUTKA, K. & JARVINEN, H.M. 2005. A study of the difficulties of novice programmers. *ACM SIGCSE bulletin*, 37(3):14–18.

LEE, Y. 2011. Scratch: multimedia programming environment for young gifted learners. *Gifted child today*, 34(2):26–31.

MACKENZIE, N. & KNIPE, S. 2006. Research dilemmas: paradigms, methods and methodology. *Issues in educational research*, 16(2):193–205.

MALAN, D.J. & LEITNER, H.H. 2007. Scratch for budding computer scientists. *ACM SIGCSE bulletin*, 39(1):223–227.

MALONEY, J., PEPPLER, K., KAFAI, Y., RESNICK, M. & RUSK, N. 2008. Programming by choice: urban youth learning programming with Scratch. *ACM SIGCSE bulletin*, 40(1):367–371.

MALONEY, J., RESNICK, M., RUSK, N., SILVERMAN, B. & EASTMOND, E. 2010. The Scratch programming language environment. *ACM transactions on computing education*, 10(4):1–16.

MAREE, K. & VAN DER WESTHUIZEN, C. 2007. Planning a research proposal. (*In* Maree, K., *ed*. First steps in research. Pretoria: Van Schaik. p. 24–45.)

MEERBAUM-SALANT, O., ARMONI, M. & MORDECHAI, B.A. 2013. Learning computer science concepts with Scratch. *Computer science education*, 23(3):239–245.

MEERBAUM-SALANT, O., MICHAL, A. & BEN-ARI, M. 2011. Habits of programming in Scratch. (*In* Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, Darmstadt. NY: ACM. p. 168–172.)

MENTZ, E. & GOOSEN, L. 2007. Are groups working in the Information Technology class? *South African journal of education*, 27(2):329–343.

MENTZ, E., VAN DER WALT, J.L. & GOOSEN, L. 2008. The effect of incorporating cooperative learning principles in pair programming for student teachers. *Computer science education*, 18(4):247–260.

MERRIAM, S.B. 1998. Qualitative research and case study applications in education. San Francisco, CA: Jossey-Bass. 275 p.

MEVARECH, Z.R. & KRAMARSKI, B. 1997. IMPROVE: a multidimensional method for teaching mathematics in heterogeneous classrooms. *American educational research journal*, 34(2):365–394.

MISA, T.J. 2010. An interview with Edsger W. Dijkstra. *Communcations of the ACM*, 53(8):41–47.

MIT MEDIA LAB. 2003. About Scratch. http://scratch.mit.edu/about/ Date of access: 3 Aug. 2013.

MIT MEDIA LAB. 2007. Scratchers' forum. http://Scratch.mit.edu/forums/viewtopic.php?id=110511 Date of access: 28 Nov. 2012.

MIT MEDIA LAB. 2014a. Saving data. http://wiki.scratch.mit.edu/wiki/saving_data Date of access: 18 Mar. 2014.

MIT MEDIA LAB. 2014b. Script. http://wiki.scratch.mit.edu/wiki/Script Date of access: 15 Oct. 2014.

MORSE, J.M., BARRETT, M., MAYAN, M., OLSON, K. & SPIERS, J. 2002. Verification strategies for establishing reliability and validity in qualitative research. *International journal of qualitative methods*, 1(2):13–22.

NIEUWENHUIS, J. 2007. Introducing qualitative research. (*In* Maree, K., *ed*. First steps in research. Pretoria: Van Schaik. p. 46–68.)

NORMAN, G.R. & SCHMIDT, H.G. 1992. The psychological basis of problem-based learning: a review of the evidence. *Academic medicine*, 67(9):557–565.

NUUTILA, E., TÖRMÄ, S. & MALMI, L. 2005. PBL and computer programming: the seven steps method with adaptations. *Computer science education*, 15(2):123–142.

OLIVIER, M.S. 2009. Information technology research. 3rd ed. Pretoria: Van Schaik.

PONTEROTTO, J.G. 2005. Qualitative research in counseling psychology: a primer on research paradigms and philosophy of Science. *Journal of counseling psychology*, 52(2):126–136.

RESNICK, M. 2012. Let's teach kids to code. http://www.ted.com/talks/mitch_resnick_let_s_teach_kids_to_code.html Date of access: 6 Mar. 2013.

RESNICK, M., MALONEY, J., MONROY-HERNANDEZ, A., RUSL, N., EASTMOND, E., BRENNAN, K., MILLER, A., ROSENBAUM, E., SILVER, J., SILVERMAN, B. & KAFAI, Y. 2009. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67.

ROBERTS, E., BRUCE, K., CUTLER, R., CROSS, J., GRISSOM, S., KLEE, K., RODGER, S., TREES, F., UTTING, I. & YELLIN, F. 2006. ACM Java task force. http://jtf.acm.org/ Date of access: 21 Mar. 2013.

ROBINS, A., ROUNTREE, J. & ROUNTREE, N. 2003. Learning and teaching programming: a review and discussion. *Computer science education*, 13(2):137–172.

ROGALSKI, J. & SAMURCAY, R. 2010. Task analysis and cognitive model as a framework to analyse environments for learning programming. (*In* Lemut, E., Du Boulay, B. & Dettori, G., *eds*. Cognitive models and intelligent environments for learning programming. Berlin: Springer-Verlag. p. 6–19.)

SAJANIEMI, J. & KUITTINEN, M. 2008. From procedures to objects: a research agenda for the psychology of object-oriented programming education. *Human technology: an interdisciplinary journal on humans in ICT environments*, 4(1):75–91.

SAMS, A. & BERGMANN, J. 2013. Flip your students' learning. *Educational leadership*, 70(6):16–20.

SANDI-URENA, S., COOPER, M.M. & STEVENS, R.H. 2010. Enhancement of metacognition use and awareness by means of a collaborative intervention. *International journal of science education*, 33(3):1–18.

SCHMIDT, H. 1983. Problem-based learning: rationale and description. *Medical education*, 17(1):11–16.

SCHNELKER, D.L. 2006. The student-as-bricoleur: making sense of research paradigms. *Teaching and teacher education*, 22(1):42–57.

SEBESTA, R.W. 2012. Concepts of programming languages. 10th ed. Harlow: Pearson.

SIEMENS, G. 2004. Connectivism: a learning theory for the digital age. *International journal of instructional technology and distance learning*, 2(1):3–10.

SMITHERMAN, C.E. & GIRARD, A.K. 2011. Creating connection: composition theory and creative writing craft in the first-year writing classroom. *Currents in teaching and learning*, 3(2):49–57.

STERNBERG, R.J. & STERNBERG, K. 2012. Cognition. 6th ed. Belmont, CA: Wadsworth.

SUMIGA, J. 2010. Programming and design. (*In* Lemut, E., Du Boulay, B. & Dettori, G., *eds*. Cognitive models and intelligent environments for learning programming. Berlin: Springer-Verlag. p. 59–69.)

TEASLEY, B.M. 2010. Program comprehension skills and their acquisition: a call for an ecological paradigm. (*In* Lemut, E., Du Boulay, B. & Dettori, G., *eds*. Cognitive models and intelligent environments for learning programming. Berlin: Springer-Verlag. p. 71–79.)

TRAYNOR, D. & GIBSON, P. 2004. Towards the development of a cognitive model of programming: a software engineering approach. Proceedings of the 16th Workshop of Psychology of Programming Interest Group, Carlow. http://www.cs.may.ie/~dtraynor/papers/PPIGarticle.pdf Date of access: 28 Aug. 2012.

UTTING, I. COOPER, S., KÖLLING, M., MALONEY, J. & RESNICK, M. 2010. Alice, Greenfoot and Scratch: a discussion. *ACM transactions on computing education*, 10(4):1–11.

VAN DER WALT, J.L. & POTGIETER, F.J. 2012. Research method in education: the frame by which the picture hangs. *International journal of multiple research approaches*, 6(3):220–232.

VYGOTSKI, L. 1963. Learning and mental development at school age. (*In* Simon, B. & Simon, J., *eds*. Educational psychology in the U.S.S.R. Stanford: Routledge and Kegan Paul. p. 21–34.)

WALSHAM, G. 1995. The emergence of interpretivism in IS research. *Information systems research*, 6(4):379–394.

WANG, X. & ZHOU, Z. 2011. The research of situational teaching mode of programming in high school with Scratch. (*In* Information Technology and Artificial Intelligence Conference (ITAIC), 6th IEEE Joint International, Vol. 2, Chongqing. IEEE. p. 488–492.)

WEISFELD, M. 2009. The object-oriented thought process. 3rd ed. Upper Saddle River, NJ: Addison-Wesley. 330 p.

WILLIAMS, B. 2004. Self direction in a problem based learning program. *Nurse education today*, 24(4):277–285.

WILLIAMS, L., WIEBE, E., YANG, K., FERZLI, M. & MILLER, C. 2002. In support of pair programming in the introductory computer science course. *Computer science education*, 12(3):197–212.

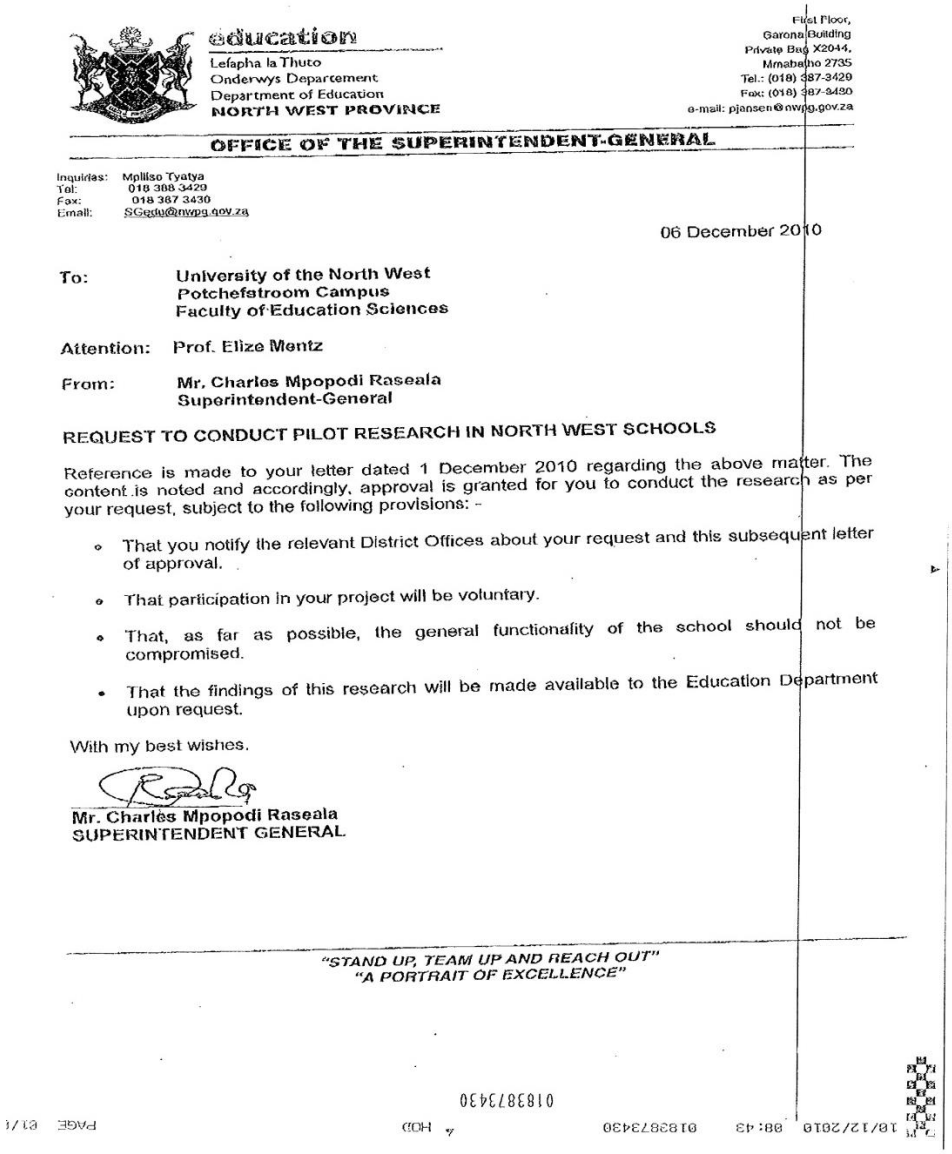
WOLZ, U., LEITNER, H.H., MALAN, D.J. & JOHN, M. 2009. Starting with Scratch in CS 1. *ACM SIGCSE bulletin*, 41(1):1–3.

WOLZ, U., MALONEY, J. & PULIMOOD, S.M. 2008. Scratch your way to introductory CS. *ACM SIGCSE bulletin*, 40(1):298–299.

YADIN, A. 2013. Improve abstract reasoning in computer introductory courses. *International journal of modern education and computer science*, 5(1):14–20.

ADDENDUM A

TOESTEMMINGSBRIEF VANAF NOORDWESPROVINSIE DEPARTEMENT VAN BASIESE ONDERWYS



education
Lefapha la Thuto
Onderwys Departement
Department of Education
NORTH WEST PROVINCE

Fifet Floor,
Garona Building
Private Bag X2044,
Mmabatho 2735
Tel.: (018) 387-3429
Fax: (018) 387-3490
e-mail: pjansen@nwpg.gov.za

OFFICE OF THE SUPERINTENDENT-GENERAL

Inquiries: Mpiliso Tyatya
Tel: 018 388 3429
Fax: 018 387 3430
Email: SGedu@nwpg.gov.za

06 December 2010

To: University of the North West
Potchefstroom Campus
Faculty of Education Sciences

Attention: Prof. Elize Mentz

From: Mr. Charles Mpopodi Raseala
Superintendent-General

REQUEST TO CONDUCT PILOT RESEARCH IN NORTH WEST SCHOOLS

Reference is made to your letter dated 1 December 2010 regarding the above matter. The content is noted and accordingly, approval is granted for you to conduct the research as per your request, subject to the following provisions: -

- That you notify the relevant District Offices about your request and this subsequent letter of approval.
- That participation in your project will be voluntary.
- That, as far as possible, the general functionality of the school should not be compromised.
- That the findings of this research will be made available to the Education Department upon request.

With my best wishes.

Mr. Charles Mpopodi Raseala
SUPERINTENDENT GENERAL

"STAND UP, TEAM UP AND REACH OUT"
"A PORTRAIT OF EXCELLENCE"

ADDENDUM B

BRIEF AAN SKOOLHOOF

NORTH-WEST UNIVERSITY ®
YUNIBESITI YA BOKONE-BOPHIRIMA
NOORDWES-UNIVERSITEIT
INSTITUSIONELE KANTOOR

Privaat sak X1290, Potchefstroom
Suid-Afrika 2520Web: http://www.nwu.ac.za
Rekenaarwetenskaponderwys
Tel: 018 2991472
Faks: 018 2994238
Sel: 082 574 6046
E-pos: 11053836@nwu.ac.za

21 Januarie 2013

Die Skoolhoof

Hoërskool _____

Geagte

NAVORSING: Die onderrig van *Scratch* in die vak Inligtingstegnologie

Ek is tans besig met my Meestersgraad in Rekenaarwetenskaponderwys en doen navorsing oor die onderrig van *Scratch* in graad 10 met die oog op die oorgang na *Delphi* in graad 11, en die bemagtiging van IT-onderwysers daarvoor. My studie is onder leiding van Prof Elsa Mentz as deel van ’n groter South African Netherlands Research Program on Alternative Development (SANPAD) projek. Die Noordwes Departement van Onderwys het reeds toestemming verleen vir hierdie navorsing in skole in die provinsie.

Hiermee versoek ek u vriendelik om toestemming te verleen dat u IT-onderwyser betrek kan word by hierdie navorsing. Dit behels twee onderhoude met u IT-onderwyser van ongeveer ’n halfuur elk oor die onderrig van *Scratch* en *Delphi*. Die onderhoude sal nie inbreuk maak op leerders se onderrigtyd nie. Indien u skool bereid is om aan die navorsing deel te neem, sal ek self ’n afspraak met die onderwyser maak op ’n tyd wat hom/haar pas. Die bevindings sal aan die IT-onderwyser beskikbaar gestel word.

Indien u enige verdere inligting benodig, is u welkom om my te skakel by die telefoonnommers hierbo. Ek sal dit verder waardeer indien u die aangehegte besonderhede aan my kan terug faks of per e-pos kan terugstuur.

Vriendelike groete

Mev. Sukie van Zyl (Junior Lektor)
Vakgroep Rekenaarwetenskap-onderwys

ADDENDUM C

TOESTEMMINGSBRIEF VANAF SKOOLHOOF

NORTH-WEST UNIVERSITY
YUNIBESITI YA BOKONE-BOPHIRIMA
NOORDWES-UNIVERSITEIT
POTCHEFSTROOM CAMPUS

Privaat sak X1290, Potchefstroom, Suid-Afrika 2520

Rekenaarwetenskaponderwys

Tel: 018 2991472

Faks: 018 2994238

Sel: 082 574 6046

E-pos: 11053836@nwu.ac.za

1 Februarie 2013

NAVORSING: Die onderrig van *Scratch* in die vak Inligtingstegnologie

Voltooi assblief en faks/e-pos terug na:
Mev Sukie van Zyl
FAKS: 018 299 4238 / E-POS: 11053836@nwu.ac.za

Die Hoof: Hoërskool _____

☐ Ek gee hiermee toestemming dat die navorsing in my skool uitgevoer mag word

☐ Ek versoek dat die navorser my kontak en meer inligting verskaf

ENIGE ANDER OPMERKINGS OF VERSOEKE:

_____

_____

_____ _____

HANDTEKENING HOOF DATUM

GR 10 IT-ONDERWYSER:_____

KONTAKNOMMER VAN GR 10 IT-ONDERWYSER:_____

☐ Ek is gewillig om aan die navorsing deel te neem

ENIGE ANDER OPMERKINGS OF VERSOEKE:

_____

_____

_____ _____

HANDTEKENING: GR 10 IT-ONDERWYSER DATUM

ADDENDUM D

ONDERHOUDPROTOKOL EERSTE ONDERHOUDE

Onderhoudprotokol: Eerste onderhoud Die onderrig van *Scratch*		
Tyd:	Datum:	Plek:
Beskryf projek en die doel van die studie, wat gedoen gaan word om vertroulikheid van data te verseker en hoe lank onderhoud gaan neem.		
Deelnemer lees en teken toestemmingsbrief. Sit bandopnemers aan.		
Vraag 1: Verduidelik wat u doel was met die onderrig van *Scratch*.		
Vraag 2: Verduidelik watter onderrig–leerstrategie u gebruik het om *Scratch* te onderrig.		
Vraag 3: Verduidelik watter spesifieke programmeringsbeginsels/-begrippe u met *Scratch* onderrig het en hoe u dit gedoen het.		
Vraag 4: Hoe lank het u *Scratch* in graad 10 onderrig voordat u met *Delphi* begin het?		
Bedank deelnemer vir tyd en bereidwilligheid om deel te neem. Vra of deelnemer enige vrae het. Verseker deelnemer van vertroulikheid.		

ADDENDUM E

ONDERHOUDPROTOKOL TWEEDE ONDERHOUDE

Onderhoudprotokol: Tweede onderhoud Ervaring van oorgang na *Delphi*		
Tyd:	Datum:	Plek:
Beskryf projek en die doel van die studie, wat gedoen gaan word om vertroulikheid van data te verseker en hoe lank onderhoud min of meer gaan neem.		
Sit bandopnemers aan.		
Vraag 1: Hoe beleef graad 11 leerders die oorgang van *Scratch* na *Delphi*?		
Vraag 2: Watter afdelings van die werk of begrippe van programmering in *Delphi* het u al met die leerders behandel?		
Vraag 3: Hoe vergelyk leerders se begrip van *Delphi*-programmering met die vorige jare toe *Scratch* nie in graad 10 onderrig is nie?		
Vraag 4: Waarmee sukkel die leerders in *Delphi* en is daar begrippe wat hulle maklik vind?		
Vraag 5: Het u tendense waargeneem waar leerders heeltyd dieselfde foute maak of swak programmeringstegnieke toepas?		
Vraag 6: Kan u enige opmerkings maak oor die invloed van *Scratch* op die aanleer van *Delphi*?		

ADDENDUM F

PROGRAMMERINGSBEGRIPPE WAT VOLGENS DIE KABV REEDS IN GRAAD 10 ONDERRIG MOET WORD

Programmeringsbegrip	Beskrywing
Algoritmes	• Alledaagse voorbeelde van – ontwerp, gebruik, interpreteer
Aanleer van probleemoplossingsprosedures en verkenning en ontwikkeling van algoritmes	• Bepaal kleinste en grootste waardes • Omruil van waardes • Aggregaatbepaling • Basiese berekenings • Bepaal of ’n getal ewe is • Bepaal of ’n getal ’n faktor van ’n ander getal is • Bepaal of getal ’n priemgetal is • Bereken grootste gemene deler en kleinste gemene veelvoud
Hulpmiddels vir voorstelling van algoritmes	• Vloeidiagramme • Pseudokode
Toetsing van algoritmes en voorstelling daarvan in programmeringskode	• Gebruik naspeurtabel om korrektheid te bepaal • Vergelyk algoritmes ten opsigte van volgorde, akkuraatheid en effektiwiteit • Verstaan die waarde van akkurate, goed getoetste algoritmes
Veranderlikes	• Verken die gebruik daarvan • Globale vs. lokale veranderlikes • Benoemingskonvensies • Toekenning van waardes • Datatipes: heelgetal, string, reëel, Boole
Operators	• +,-,*,/ • Volgorde van bewerking • Vergelykingsoperators, Boole-operators en die uitvoer van logiese bewerkings • Bepaal die res, modulus
Stringe	• Stringoperators en stringbewerkings • Bepaal spesifieke karakter in ’n string
Funksies	• Ewekansig, afrond, vierkantswortel
Berekenings	• Oppervlak, volume en eenvoudige formules en berekenings wat gewoonlik in ander vakke gedoen word

Programmeringsbegrip	Beskrywing
Besluitneming	• IF en IF-THEN-ELSE
Herhaling/Iterasie	• FOR, pre-voorwaardelike en post-voorwaardelike herhalingstrukture (REPEAT, REPEAT UNTIL en FOREVER)
Gebeurtenishantering	• When clicked, When keypressed, Broadcast en When I receive
Basiese valideringstegnieke	• Valideer toevoer en verwerking • Toets vir deling deur nul, negatiewe vierkantswortel
Ontfoutingstegnieke	• Ontfouting deur gebruik van die dophou fasiliteit
Lyste/begrippe rakende skikkings	• Stoor van en toegang tot ŉ lys van getalle en stringe • Manipuleer van lyste, soos byvoeg, uithaal, vervang, invoeg van items • Parallelle lyste • Vind ŉ gegewe item binne ŉ lys • Verken eenvoudige, geneste herhalingstrukture

ADDENDUM G

SERTIFIKAAT VAN TAALVERSORGING

Jackie Viljoen
Algemene Taalpraktisyn
Bergzicht Gardens 16
Fijnbosslot
STRAND 7140

Geakkrediteerde lid van die Suid-Afrikaanse Vertalersinstituut
Nommer APSTrans 1000017
Lid van die Professional Editors' Group (PEG)

☎ +27+21-854 5095 082 783 0263 086 585 3740
Posadres: Bergzicht Gardens 16, Fijnbosslot, STRAND 7140, Suid-Afrika

VERKLARING

Ek verklaar hiermee dat die M-tesis van **SUKIE VAN ZYL** behoorlik deur my taalversorg is, maar sonder insae in die finale weergawe.

Titel van tesis:
DIE ONDERRIG VAN *SCRATCH* MET DIE OOG OP DIE AANLEER VAN *DELPHI* AS OBJEKGEORIËNTEERDE PROGRAMMERINGSTAAL

JACKIE VILJOEN
STRAND
Suid-Afrika
29 September 2014