Katholieke
Universiteit
Leuven

Faculteiten Wetenschappen
en Toegepaste Wetenschappen

Experimenteel programmeren in een object gericht systeem

Kris VAN HEES

Eindverhandeling aangeboden tot het
behalen van de graad van licentiaat
in de informatica

1994–1995

Promotor : Prof. Dr. ir. E. STEEGMANS

Faculteiten Wetenschappen
en Toegepaste Wetenschappen

K.U. Leuven
Academiejaar 1994–1995

Naam en voornaam student: Van Hees Kris

Titel:

Experimenteel programmeren in een object gericht systeem

Engelse vertaling:

Experimental programming in an object oriented system

Object gericht programmeren is een erg populaire methodologie in de informatica. Het blijkt echter ook een struikelblok te zijn voor vele studenten, dikwijls ten gevolge van het feit dat onderricht in dit paradigma voornamelijk theoretisch blijkt te zijn. De ervaring leert ons dat programmeertechnieken op een bijna speelse wijze aangeleerd kunnen worden door ermee te experimenteren. Vandaar het nut van een werkomgeving die dergelijke experimenten zeer goed toelaat.

Een MUD (*Multi–User Dungeon*) is een avonturenspel dat zich situeert in een tekstuele virtuele wereld. Talrijke gebruikers spelen tegelijkertijd en treden met elkaar in interactie. Het belangrijkste aspect uit oogpunt van deze thesis is wel het feit dat gebruikers de spelwereld kunnen uitbreiden door middel van object gericht programmeren in een ingebouwde programmeertaal.

Vertrekkend uit de kennis die MUDs geven betreffende het gemeenschappelijk leerproces van gebruikers wordt onderzocht in hoeverre de taal die in de MUD gebruikt wordt ook bruikbaar is voor onderricht in object gericht programmeren. Verscheidene aanpassingen moeten gebeuren aan de programmeertaal, en ook moet er een degelijke programmeeromgeving opgebouwd worden die de voordelen van een MUD combineert met de noden van een educatieve toepassing. Deze thesis implementeert een aangepaste versie van de programmeertaal LPC en maakt tevens een prototype van een bijhorende programmeeromgeving op basis van deze taal. Eén van de voorbeelden die in dit prototype aanwezig zijn is de implementatie van een beperkte World Wide Web server, die tevens een grafische interface verzorgt voor de programmeeromgeving.

Eindverhandeling aangeboden tot het behalen
van de graad van licentiaat in de informatica

Promotor:	Prof. Dr. ir. E. Steegmans	Departement Computerwetenschappen
Assessoren:	Prof. ir. J. Lewi	
	Prof. Dr. ir. Y. Berbers	

Inhoudsopgave

1 Inleiding **1**

1.1 Uitgangspunt . . . 2

1.2 Doel van de thesis . . . 2

1.3 Overzicht . . . 5

1.4 Dank. 5

2 De taal LPC **6**

2.1 Historiek van LPmud . . . 6

2.1.1 LPmud 1.x . . . 7

2.1.2 LPmud 2.x . . . 7

2.1.3 LPmud 3.x . . . 9

2.1.4 LPmud 4.x? . . . 10

2.2 Basisconcepten . . . 11

2.2.1 Datatypes . . . 11

2.2.2 Klassen en entiteiten . . . 14

2.2.3 Functieoproepen . . . 15

2.2.4 Parameterbindingen en datatypes . . . 23

2.2.5 De wijzer–type anomalie . . . 27

2.2.6 Inheritance . . . 28

2.2.7 Controlestructuren . . . 30

2.2.8 Waarom werden LPC en DGD gekozen? . . . 34

3 Implementatie **35**

3.1 Inleiding betreffende de werking van DGD . . . 36

3.1.1 Geheugenbeheer, swap en dump . . . 37

3.1.2 Opkuisen met verwijzingstellers . . . 38

3.1.3 Opkuisen tijdens swapping . . . 38

3.1.4 Opkuisen bij een dump . . . 39

3.2 Functieoproepen . . . 39

3.3 Klasse en entiteit concept . . . 40

3.4 Naamgeving van klassen en entiteiten . . . 42

3.4.1 Conversie van arbitraire namen naar bestandsnamen . . . 43

3.4.2 Het vertalingsproces . . . 44

3.4.3 Nevereffecten . . . 46
3.4.4 Implementatiedetails . . . 46
3.5 De wijzer–type anomalie . . . 47
3.5.1 Wijzer–types intern in DGD . . . 48
3.5.2 Eerste mogelijke implementatie . . . 50
3.5.3 Tweede mogelijke implementatie . . . 50
3.5.4 Derde mogelijke implementatie . . . 51
3.5.5 Verdere uitwerking van de implementatie . . . 53
3.5.6 Voorbeeld . . . 56
3.5.7 Een onopgelost probleem? . . . 63

4 De LPC programmeeromgeving 64
4.1 De basis van de LPC omgeving . . . 65
4.2 De toegangsprocedure: een voorbeeld . . . 66
4.3 De dialoog met de gebruiker . . . 68
4.3.1 De tekstuele dialoog . . . 69
4.3.2 De WWW dialoog (fakultatief) . . . 70
4.4 Een voorbeeld . . . 70
4.5 Ontwerpaspecten . . . 72

5 MUDs 74

6 Besluit 79
6.1 Het werk . . . 79
6.2 De doelstellingen: een terugblik . . . 80
6.3 De toekomst... . . . 81
6.4 Eindconclusie . . . 82

A LPC syntax 83

B Een aantal basis klassen en entiteiten 87
B.1 De driver entiteit: /kernel/init.c . . . 87
B.2 De naam conversie: /kernel/named.c . . . 90
B.3 De trie klasse: /core/trie.c . . . 91

Bibliografie 95

Hoofdstuk 1

Inleiding

"You can always tell a really good idea by the enemies it makes."
(Axioma voor programmeurs)

Object gerichte programmatie is reeds een hele tijd een populaire techniek in de informatica. Het principe van *object gerichtheid* wordt beschouwd als de eerste methodologie waarin analyse, ontwerp en implementatie sterk samenhangen, en gelijke principes hanteren, zonder wanordelijk in elkaar over te vloeien. Onderricht in informatica is bijna niet meer in te denken zonder enige initiatie in het object gericht paradigma. Dikwijls komt de nadruk hier echter meer te liggen op de theoretische aspecten dan op de effectieve praktijk.

Ervaring leert nochtans dat praktisch onderricht belangrijk is om inzicht te krijgen in de problematiek. De vraag die men zich dan dient te stellen is: "waarom komt de kennis die de ervaring ons geeft in dit geval onvoldoende aan bod?" Het antwoord moet vrijwel zeker gezocht worden in functie van de beschikbare hulpmiddelen. Zonder een degelijke programmeertaal en bijhorende omgeving (vertaler of vertolker, hulpprogramma's, ...) is het zo goed als onmogelijk om goed praktisch onderricht te geven. Programmeertalen zoals Smalltalk–80, Modula–3, C++ en Eiffel zijn niet zo goed, om verschillende redenen. Enerzijds is er in Smalltalk–80 en Modula–3[1] geen ondersteuning voor multiple inheritance[2]. Anderzijds is C++ te uitgebreid en te complex, en zijn er in de ontwikkeling ervan redelijk ongewone keuzen gemaakt uit oogpunt van de efficiëntie. Tenslotte geeft geen van deze talen een open, dynamische werkomgeving[3], wat in deze thesis wel het geval is. Uiteraard betekent dit niet dat alle andere talen onbruikbaar zijn. Deze thesis wil echter een alternatief aangeven dat mogelijk dichter staat bij de behoeften van het object gericht programmeren, en dan in eerste instantie voor het aanleren van deze principes.

[1] Andere opmerkingen betreffende talen die modules met objecten (of type extensies) ondersteunen in plaats van enkel objecten, worden in [HZ93] gegeven.

[2] In deze thesis wordt de term inheritance verkozen boven de nederlandse vertaling "overerving" omdat de engelstalige term meer ingeburgerd is in de informaticawereld. Dit impliceert dan de keuze multiple inheritance te gebruiken.

[3] "Open" in de zin dat alle gebruikers samen in éénzelfde omgeving werken. "Dynamisch" in de zin dat de omgeving blijft bestaan tussen experimenten, en in theorie nooit wordt herstart.

Deze observatie heeft geleid tot de conclusie dat een redelijk eenvoudige, doch krachtige programmeertaal met een degelijke omgeving kan bijdragen tot een beter onderricht in object gerichte principes. Uiteraard kan dit veralgemeend worden naar andere aspecten van de informatica, doch dit valt buiten het bestek van deze thesis.

1.1 Uitgangspunt

Uitgangspunt voor deze thesis was het fenomeen van de (meestal onterecht negatief bekeken) *MUDs*. Een MUD[4] is een implementatie van een virtuele wereld waarin meerdere gebruikers met elkaar in interactie treden. In de huidige generatie MUDs is het mogelijk die wereld verder uit te bouwen door het programmeren van objecten. Dit gebeurt niet enkel door de makers van de wereld, maar door een heleboel gebruikers, die dikwijls geen voorkennis hebben van informatica (geneeskundigen, economisten, advocaten, . . .).

Dit belangrijk aspect heeft zeer veel bijgedragen tot de aanvang van deze thesis omdat men in MUDs reeds na korte tijd (enkele weken) de object gerichte basisprincipes leert toepassen (meestal evenwel zonder te beseffen dat men één van de krachtigste technieken uit de informatica onder de knie hebben), terwijl de gemiddelde informaticastudent een hele strijd moet leveren voordat hij of zij met object gerichte technieken overweg kan. Deze licht overdreven stelling blijkt in praktijk heel wat waarheid te bevatten. De thesis die nu voor u ligt probeert de strijd van de arme student te verlichten door een werkomgeving ter beschikking te stellen die de voordelen van de MUD omgeving combineert met de vereisten van een goed praktisch onderricht.

Als basis voor de thesis werd één bepaalde implementatie gekozen, namelijk DGD[5] dat de programmeertaal LPC[6] aanbiedt aan de gebruiker. DGD is een vertaler/vertolker combinatie die tevens de interactie verzorgt tussen de LPC omgeving en de gebruikers. Gebruikers communiceren met DGD door middel van netwerkverbindingen (bijvoorbeeld "telnet" of een gespecialiseerd programma).

1.2 Doel van de thesis

Het feitelijke doel van de thesis kan omschreven worden als:

> *De ontwikkeling van een object gericht programmeersysteem voor educatieve doeleinden.*

Deze kernachtige omschrijving is de synthese van de belangrijkste uitgangspunten die aan de grondslag liggen van dit werk. Eén na één zullen deze principes onder de loupe genomen worden.

[4]MUD is een acroniem voor Multi–User Dungeon.

[5]DGD staat voor "Dworkin's Generic Driver". Het programma werd ontworpen en geschreven door Felix A. Croes (in MUDs beter gekend als Dworkin).

[6]LPC staat voor Lars Pensjö C. Lars schreef de allereerste implementatie en heeft deze lange tijd blijven herwerken. Momenteel is hij niet meer aktief betrokken bij het MUD gebeuren.

- **Ontwikkeling**
  Allereerst betreft het hier een ontwikkeling. Dit betekent dat er een evolutie is die moet leiden tot een vooropgesteld doel. Het is niet de bedoeling een puur theoretische verhandeling te maken. Op basis van een theoretische studie wordt een praktische implementatie uitgewerkt.

- **Object gericht**
  Verder hebben we te maken met een object gericht aspect. Dit is een redelijk delicaat punt, daar er geen éénduidige definitie bestaat voor deze term. Heel wat pogingen zijn ondernomen om tot een algemeen aanvaardbare definitie te komen [Cop92, MO92, R+91, SM88], doch zonder succes. Het lijkt dan ook aangewezen een definitie te geven die geldig is voor deze thesis.

  1 *Een object is een abstractie van de reële wereld met een interface van functies en meestal ook een verborgen toestand.*

  2 *Gemeenschappelijke eigenschappen tussen objecten wordt bekomen door middel van inheritance.*

  3 *Objecten die enkel kunnen verschillen in toestand en niet in gedrag worden bekomen als klonen van een voorbeeld–object, dat we dan een klasse noemen.*

  Deze definitie zal zeker niet voor iedereen in zijn geheel aanvaardbaar zijn. Vooral deel 1 blijkt een punt van discussie te bevatten. Er is namelijk een grote groep informatici die van mening zijn dat een object een abstract datatype moet zijn. Objecten met uitsluitend een set van functies, en geen toestand zijn volgens hen ontoelaatbaar. Bertrand Meyer lijkt deze mening ook aanhankelijk te zijn, op basis van de definitie die hij geeft in [Mey92, pagina 4], doch iets verder vermeldt hij "*klassen die slechts nuttig zijn als een module die een aantal functies omsluit*" ([Mey92, pagina 44]). Dit is duidelijk een vermelding van klassen (of objecten) die geen abstract datatype beschrijven. Daar dit een algemenere visie is, en het nut van dergelijke functionele klassen groot kan zijn, wordt in deze thesis de meer algemene visie gebruikt.

  De rest van de definitie komt overeen met de algemene eigenschappen die men meestal in object gerichte middens als *normaal* beschouwt.

  Als men deze definitie toetst aan de criteria die Grady Booch in [Boo94] aangeeft, dan is het duidelijk dat de vier hoofdkenmerken aanwezig zijn: abstractie, encapsulatie, modulariteit en hiërarchie. De nevenkenmerken typering, concurrentie en persistentie zijn niet of slechts gedeeltelijk aanwezig. Je kan typering namelijk expliciet implementeren door een gepaste herdefinitie van functieoproep faciliteiten, en persistentie is eigenlijk deels aanwezig in de vorm van basisfuncties die toelaten de toestand van een object naar een willekeurig bestand te schrijven en die terug in te lezen. In de beschrijving van de klasse kan aangegeven worden welke variabelen[7]

[7] Men noemt dit ook wel attributen van een klasse.

niet weggeschreven hoeven te worden[8] (lokale variabelen in functies worden nooit weggeschreven.

- **Programmeersysteem**
  Het is de bedoeling uiteindelijk een programmeersysteem te bekomen. Dit betekent dat niet enkel een programmeertaal nodig is, maar ook al de hulpmiddelen om met deze taal te werken. Dergelijke hulpmiddelen omvatten een geheel van standaard klassen, voorzieningen om programmatekst–manipulaties uit te voeren, een vertaler/vertolker, ...

  Binnen het kader van de thesis wordt *geen* nieuwe taal ontwikkeld. Wel wordt er een dialect gebruikt van een bestaande taal. Uitgangspunt is de taal LPC zoals deze in DGD gedefiniëerd wordt. Op basis van de doelstellingen die in dit hoofdstuk worden gegeven, zijn er een aantal aanpassingen gebeurd aan de taal. Specifieke details betreffende de taal en aanpassingen worden beschreven in het volgende hoofdstuk. Om tot een werkbaar geheel te komen zijn nog een aantal hulpmiddelen nodig, en deze worden ook ontwikkeld in functie van deze thesis. Wat er zoal nodig is, werd in de vorige paragraaf opgesomd.

  Concreet wordt er dus een bestaand programma (DGD) zodanig aangepast dat het voldoet aan de gestelde eisen, en om een meer consistente object gerichte programmeertaal aan te bieden aan de gebruiker. Verder wordt er een geheel van klassen uitgewerkt dat de werkomgeving voor de gebruiker zal vormen, dit op een flexibele manier om latere uitbreidingen met een gebruikersprogramma toe te laten. Ook worden er voorbeeldklassen gemaakt om referentiemateriaal te bieden voor een beginnend gebruiker, en er zullen faciliteiten uitgewerkt worden om het werken aangenamer te maken (communicatie tussen gebruikers, help functies, ...)

- **Educatieve doeleinden**
  De functie die het programmeersysteem moet kunnen vervullen, valt onder de educatieve doeleinden. Geïnteresseerden moeten in staat zijn het ontwikkelde systeem te gebruiken om vertrouwd te raken met object gericht programmeren. Niet door theoretische studie, maar op basis van zelf uitgevoerde experimenten.

Uiteraard is de uitwerking van deze doelstellingen tot een mooi afgewerkt geheel een immense taak, en zeker teveel werk voor een thesis. De bedoeling is dan ook een prototype te ontwikkelen dat in een later stadium afgewerkt kan worden tot een algemeen bruikbaar eindprodukt.

---

[8] Zie hiervoor pagina 13.

1.3 Overzicht

De basis van een MUD is een programma[9] dat de uitvoering verzorgt van de functies die aanwezig zijn in de objecten die de wereld voorstellen. In deze thesis wordt met een aangepaste versie van LPC gewerkt omdat de standaard implementatie niet voldoet voor een educatieve omgeving. Ook zijn er een aantal onzuiverheden in de taal die niet aanvaardbaar zijn.

In hoofdstuk 2 worden de meest belangrijke concepten van de taal LPC beschreven. Hier wordt ook een bespreking gegeven van de fouten in de standaard implementatie en de mogelijke oplossingen. Op het einde van dit hoofdstuk wordt besproken waarom nu net LPC werd gekozen als taal, in functie van de doelstellingen en in functie van de concepten die in de taal aanwezig zijn.

Uit de oplossingen die worden voorgesteld, wordt de beste gekozen en verder uitgewerkt in hoofdstuk 3. De belangrijkste aspecten van de implementatie worden besproken alsook de problemen die erbij optreden. Verder wordt er informatie gegeven betreffende DGD om de draagwijdte van de aanpassingen beter aan te geven.

Om een beter inzicht te geven in het verschijnsel MUD, wordt er in hoofdstuk 5 een korte illustratie gegeven van deze toch wel unieke netwerktoepassing. Het is gebaseerd op MUDs die in LPC geprogrammeerd zijn op basis van DGD, net zoals de programmeeromgeving voor deze thesis. Een drietal voorbeelden zullen illustreren hoe men zich een MUD best kan voorstellen.

Bijlage A geeft de syntax beschrijving van LPC zoals deze in de thesis gebruikt wordt.

1.4 Dank...

Deze thesis was niet mogelijk geweest zonder de steun van Prof. Steegmans, die bereid was een voor hem vrijwel onbekend gebied te betreden. De vele discussies betreffende de programmeertaal LPC waren steeds een grote hulp. Felix A. Croes is als auteur van DGD zeer zeker een belangrijke steun geweest voor deze thesis en de verschillen in visie zijn zeer verrijkend gebleken.

Dank ook aan vele MUD spelers die me tijdens de afgelopen 4 jaren getoond hebben dat men geen informaticastudent moet zijn om met object gerichte principes te werken. In het bijzonder dank aan Jonas Carlsson, Carl Christer Boräng, Anders B. Jansson, en Haijo Schipper.

Tenslotte wil ik mijn dank betuigen aan Judie Boyd, en aan mijn ouders, voor hun liefde en steun.

[9] Dit programma noemt men meestal een “gamedriver” of kortweg “driver”. In deze tekst wordt gekozen voor de meer algemene term “server” die een beter verband legt met het client–server principe dat in de informaticawereld bekend is.

Hoofdstuk 2

De taal LPC

"The three most dangerous things in the world are a programmer with a soldering iron, a hardware type with a program patch, and a user with an idea."
(Gezegde)

Een bespreking van een programmeertaal die enige evolutie heeft meegemaakt, kan niet zonder een deel van de historiek. In de evolutie van de taal is namelijk bijna altijd terug te vinden waarom bepaalde concepten al dan niet worden toegevoegd. Vertrekkend vanuit een kort historisch overzicht[1] zullen we dan ook stelselmatig een reeks concepten bespreken. Een aantal aanpassingen aan de taal LPC zullen hier aangegeven worden, om dan verder uitgewerkt te worden met de implementatiedetails in hoofdstuk 4.

2.1 Historiek van LPmud

Als men op zoek gaat naar het ontstaan van LPC, dan moet men in feite de historie van LPmud[2] bestuderen. In 1989 schreef Lars Pensjö zijn eerste versie van LPmud. Hij schrijft:

> In the beginning, I played a lot of Abermud and some Tinymud, and wanted to do something better, combining the two systems. I made the first version of LPmud as some kind of argument, to show that my ideas were possible. Luckily, I didn't know at that time how that would impact my near future.

De vermelding van Abermud[3] is voor ons van minder belang, daar het voornamelijk betrekking heeft op de spelaspecten van LPmud. Van belang is echter de Tinymud[4] referentie. Tinymud was namelijk zowat de eerste MUD waarin spelers objecten konden

[1]Deze en volgende paragrafen zijn gebaseerd op [P+93].

[2]LPmud is de naam die gegeven wordt aan MUDs die gebruik maken van de implementatie die Lars zelf schreef, of van een implementatie die erop gebaseerd is. Er is veel verwarring op dit punt, omdat sommige auteurs LPmud gebruiken als benaming voor MUDs die volgens het principe van de originele LPmud werken. Het verschil ligt in het feit dat die auteurs spreken over de LPC objecten die gelijk zijn, en niet de onderliggende taal.

[3]Abermud wordt beschreven in [A+89].

[4]Tinymud wordt beschreven in [SW90].

maken. Programmeren was nog niet mogelijk in dit spel, maar het concept van objecten die de wereld beschrijven en die door spelers gemaakt en aangepast kunnen worden, was wel aanwezig.

De vernieuwing die Lars heeft ingevoerd betreft de volledige mogelijkheid om te programmeren. Objecten in Abermud en Tinymud konden wel aangepast worden voor wat betreft beschrijvingen en berichten die gegenereerd werden, doch het gedrag van een object lag volledig vast. Deze limitering was ongewenst volgens Lars, en hij heeft gelijk gekregen.

2.1.1 LPmud 1.x

De allereerste implementatie van LPC kende nog geen inheritance, en dus moest elk object ofwel een instantiatie zijn van een bestaande klasse (een zogenaamde *kloon*), ofwel een geheel nieuw stuk code. De taal leek erg veel op een klassieke programmeertaal, met een aantal object gerichte aspecten.

2.1.2 LPmud 2.x

In 1990 kwam LPmud 2.x beschikbaar. Dit was tevens een herwerking van de taal LPC. De voornaamste vernieuwing was het invoeren van inheritance. Het ergste neveneffect was een vreselijke verwarring in terminologie tussen LPC gebruikers en *echte* programmeurs (om de toenmalige benamingen te gebruiken). Tabel 2.1 geeft een aantal van de verschillen in terminologie. Het is duidelijk dat spreken over *een object* tot hopeloze situaties leidt.

Informatica	**LPC**
Klasse	Object
Uniek object (slechts één exemplaar)	Object
Object Instantiatie	Kloon
Attribuut	Globale variabele
Methode	Functie

Tabel 2.1: Verschillen in terminologie

Gelukkig heeft de spraakverwarring geen negatieve gevolgen gehad. De MUD wereld bleef rustig haar eigen terminologie gebruiken, en uiteraard deed de informaticawereld hetzelfde. Het grootste probleem bleek echter te zitten in de conceptuele visie van de gebruikers. Veel van de standaard klassen[5] werden geschreven als een soort blauwdrukken van objecten, en dus was het gebruik ervan eerder beperkt tot het invullen van een aantal attributen in plaats van echt de mogelijkheden van inheritance toe te passen. Figuur 2.1 toont hoe code er gewoonlijk uitzag. Zie Tabel 2.2 voor een verklaring van het voorbeeld.

[5]Deze thesis gebruikt de informaticatermen, tenzij verwarring onmogelijk is.

```
 1:  inherit "obj/weapon";
 2:  reset(arg) {
 3:      if (!arg) {
 4:          set_name("knife");
 5:          set_short("a small knife");
 6:          set_long("Fairly short, but also fairly sharp.  " +
 7:                   "It is very fairly.\n");
 8:          set_class(5);
 9:          set_value(8 + random(5));
10:          set_weight(1);
11:      }
12:  }
```

Figuur 2.1: Een voorbeeld object uit LPmud 2.x

Deze figuur geeft tevens ook een tweede zeer belangrijk aspect aan: het ontbreken van datatypes. De definitie van reset()[6] geeft geen informatie over een mogelijk resultaat van de functie, en evenzeer geen informatie over het datatype van *arg*. Het was echter wel mogelijk datatypes op te geven. De vertaler/vertolker hield er gewoon geen rekening mee.

Lijn	Verklaring
1	We maken een specialisatie van de klasse die in het bestand “obj/knife” te vinden is.
2	Declaratie van functie “reset” met één argument.
3	Voer het instructieblok dat volgt enkel uit indien “arg” de waarde 0 heeft.
4—10	Dit zijn allemaal functieoproepen naar functies die in de superklasse uit bestand “obj/knife” gedefiniëerd zijn.
6	Een optelling van twee stukken tekst leidt tot een nieuw stuk tekst waarin beiden gewoon na elkaar geplaatst worden.
3 en 11	Een instructieblok wordt tussen accolades geplaatst.
2 en 12	De instructies die het gedrag van een functie beschrijven vormen een instructieblok.

Tabel 2.2: Verklaring van het voorbeeld object

[6]Per conventie zal foo() geschreven worden indien foo een functie is.

2.1.3 LPmud 3.x

Met de verschijning van LPmud 3.x kwam er een ware revolutie tot stand, die deels positief was, en deels negatief. Voor wat betreft de taal LPC kan men spreken van een serieuze verbetering. Inheritance werd aangepast om multiple inheritance toe te laten, wat de programmatuur zeker ten goede kwam (eindelijk ging men echt gebruik maken van de mogelijkheden die inheritance biedt).

Een andere belangrijke verbetering was het invoeren van een nieuw datatype, nl. *mappings*. Een mapping is in feite een associatief array[7]. Dit betekent dat in plaats van indexering via een volgnummer, indexering via een *sleutel* gebeurt. Figuur 2.2 toont hoe men zich dit best kan voorstellen. Om efficientieredenen worden de sleutels van een mapping volgens alfabetische orde bijgehouden, om snellere toegang mogelijk te maken. Dit is een gegeven dat vastligt in LPC, en dus mag men erop steunen (wat dan ook zeer dikwijls gedaan wordt, zoals uit voorbeelden in deze thesis zal volgen).

Array

Element
foo
bar
baz
apa
bepa
fnurt

Mapping

Sleutel	Waarde
Fifth	bepa
First	foo
Fourth	apa
Second	bar
Sixth	fnurt
Third	baz

Figuur 2.2: Array vs mapping

De keerzijde van de medaille was echter het ontstaan van vele dialecten, en onafhankelijke implementaties die duidelijk te leiden hadden onder het gebrek van een "standaard" LPC. Velen (zoals ook de auteur) waren van mening dat de taal zoals deze door Lars Pensjö was geïmplementeerd als standaard moest beschouwd worden. Vooral in de Verenigde Staten bleek hier discussie over te bestaan, wat leidde tot afscheuringen (zoals het MudOS systeem[8]).

Een ander negatief punt was het invoeren van een zeer groot aantal functies in de server die er eigenlijk helemaal niet thuishoren. Al wat een beetje algemeen werd beschouwd, werd in de server opgenomen. (Dit zelfde fenomeen is ook opgedoken bij bvb. het Unix besturingssysteem, en de functiebibliotheek voor de taal C.) Om een voorbeeld te nemen: in het eerder vermelde MudOS zijn er in de meest beperkte versie (geen extra faciliteiten) 162 basisfuncties. Heel wat van deze functies zijn zeer specifiek voor MUDs, en er zijn zelfs basisfuncties voor het berekenen van een CRC[9] van een tekst, allerlei filters voor arrays en

[7]Een afdoende term werd in het nederlands niet gevonden. Een "genummerde lijst" klinkt erg kunstmatig, en het lijkt dan ook best de engelstalige term over te nemen.

[8]MudOS wordt beschreven in de documentatie die te vinden is in [Kay92].

[9]Een Cyclic Redundancy Code wordt voornamelijk gebruikt om aanpassingen aan een tekst te kunnen detecteren.

mappings, ... Veelal functies die zelden of nooit nodig zijn, en die net zo goed in LPC geïmplementeerd kunnen worden. Ter vergelijking: DGD heeft slechts 51 basisfuncties (in feite zijn er 126, maar hiervan zijn er 53 operatoren die in MudOS ook niet werden meegeteld, en ook nog eens 22 wiskundige functies die evenzeer in MudOS niet werden meegeteld).

2.1.4 LPmud 4.x?

Er is reeds een versie van LPmud die verder gaat dan wat in de vorige paragrafen werd besproken. Zo is er bijvoorbeeld het LPC4 project op de universiteit van Linköping, Zweden. Hier gebruikt men een vernieuwde LPmud met een uitbreiding op LPC onder meer voor een WWW[10] server, genaamd Spider[11]. Dit is tevens een mooi voorbeeld van de mogelijkheden van LPC buiten de spelwereld.

Een andere vernieuwing die reeds enige tijd bestaat, is het concept van de zogenaamde *closures*[12]. Gezien echter de sterke syntactische banden tussen C en LPC lijkt het invoeren van deze closures eerder een rariteit dan een verbetering. Enerzijds is een groot deel van de MUD–wereld het erover eens dat het niet “mooi” is, en anderzijds past dit concept niet goed in een taal die strict genomen eigenlijk vertaald uitgevoerd wordt, en niet vertolkt wordt[13]. Er is dan ook geen consensus over een mogelijke LPmud 4.x, en het lijkt me vrijwel uitgesloten dat een versie 4 er ooit zal komen. De huidige situatie heeft een aantal implementaties die op redelijk grote schaal gebruikt worden, en elk een degelijke ondersteuning krijgen, net zoals de originele LPmud ooit de steun had van Lars Pensjö. Hoewel enige coördinatie bestaat tussen de mensen die de verschillende implementaties onderhouden, zijn de conceptuele verschillen te groot om opgelost te worden zonder gelijkvormigheid te verliezen. En aangezien er veel gebruik gemaakt wordt van al de implementaties, is gelijkvormigheid zeer zeker iets dat in gedachten gehouden wordt. Net zoals bij vele dingen leidt dit tot verstarring, waar enkel echte vernieuwingen doorheen weten te breken.

DGD is een hele vernieuwing doordat Felix Croes een minimaliteitsprincipe als uitgangspunt heeft genomen. De server bevat uitsluitend de meest noodzakelijke basisfuncties, en zelfs de interne vertolker is erg RISC[14]–gebaseerd. Dit leidt tot een zeer modulaire opbouw, en een sterk gestroomlijnde server. Het geheel is klein, en sterk genoeg om te concurreren met de andere MUD servers. Het is zelfs zo dat testen hebben aangetoond dat in vele gevallen DGD efficiënter is betreffende snelheid en geheugengebruik. En toch is veel van de code die wordt uitgevoerd, ofwel vertolkte LPC code, ofwel LPC code die door een speciale

[10]WWW staat voor World Wide Web. Dit is een zogenaamd hypertext systeem dat werd ontwikkeld in CERN.

[11]Voor meer informatie kan via World Wide Web contact gelegd worden met deze server zelf via http://www.lysator.liu.se:/.

[12]Als mogelijke nederlandse vertaling wordt soms de term “sluitingen” voorgesteld, doch dit lijkt niet erg duidelijk, dus zal in deze tekst closures gebruikt worden.

[13]Feit dat intern tot nog toe altijd met een tussentaal en een vertolker wordt gewerkt, is geen tegenargument. De vertaling naar de tussentaal zorgt voor een effectieve scheiding tussen broncode en uitvoerbare code.

[14]RISC staat voor Reduced Instruction Set Computer.

tussen–vertaler naar C werd vertaald en zo in de server werd opgenomen. Dit laatste lijkt misschien hetzelfde als het overdreven toevoegen van functies, doch dit is niet het geval. De vertaalde LPC code wordt niet opgenomen als basisfuncties maar gewoon als klassen die efficiënter behandeld worden.

2.2 Basisconcepten

Voor elke programmeertaal geldt dat kennis van de basisbegrippen uitermate belangrijk is. LPC is een nogal onbekende taal, en heeft een aantal aspecten die enige uitleg nodig hebben. Kennis van een programmeertaal als C of C++ zal zeker helpen bij een studie van LPC.

In deze kritische bespreking van LPC wordt melding gemaakt van *klassieke* talen. Dit omsluit talen van de Algol–familie, zoals onder meer Pascal, C en Algol. Een tweede klasse van talen die beschouwd wordt, is de klasse van de *object gerichte* talen, zoals Smalltalk–80, Eiffel, Self, Sather, C++ en LPC.

2.2.1 Datatypes

Datatypes zijn een veelbesproken onderwerp in object gerichtheid. Sommigen zijn voorstanders van typeloosheid, terwijl anderen vinden dat een strict systeem van datatypes nodig is. Het is natuurlijk zo dat in een goedgeschreven, correct programma datatypes niet nodig zijn. De structuur en code van het programma bepalen zelf de juiste datatypes. Spijtig genoeg is dit theorie, en niet praktijk. Programmeurs schrijven zelden foutloze programmas. Zodoende is het bestaan van datatypes eerder een hulpmiddel dan een noodzakelijkheid. Het bestaan ervan laat toe tijdens compilatie reeds fouten in verband met datatypes op te sporen en aan te geven. Voor vertolkers gebeurt deze controle dan tijdens de uitvoering, wat soms een kleine vertraging geeft, doch het voordeel van de controle is veel groter.

Voor wat betreft datatypes is LPC een nogal verouderde, onzuivere taal. Een taal zoals Smalltalk–80 definiëert datatypes uitsluitend via klassen, terwijl LPC de kant opgaat van Simula en C++. Er zijn namelijk een aantal ingebouwde datatypes die men klassiek zou kunnen noemen, en waarvoor er een aantal primitieve operaties aanwezig zijn in DGD (analoog aan de primitieve functies in Eiffel zoals beschreven in [Mey92]). Men moet geen optelfunctie oproepen in object 2 met als argument object 3, om als resultaat object 5 te krijgen. Het is *wel* mogelijk dit te implementeren met behulp van de primitieve functies.

In het aanleren van object gerichtheid wordt meestal eerst uitgelegd hoe functieoproepen werken tussen objecten, en dan wordt aangetoond dat de optelling dit mechanisme gebruikt. Het betekent echter dat men eerst een theoretische inleiding moet geven betreffende functieoproepen voordat er met de taal geëxperimenteerd kan worden. Een programmeur die zijn eerste stappen zet in de wereld van het object gericht programmeren heeft bijna altijd reeds kennis van een klassieke programmeertaal zoals Pascal of C, en met een taal als LPC kan deze programmeur onmiddellijk aan de slag met kleine experimenten.

Het principe van functieoproepen tussen objecten kan dan zeer makkelijk met voorbeelden worden aangeleerd, en in een later stadium kan de expressie $2 + 3$ genuanceerd worden als een operatie tussen objecten. Het is niet de bedoeling een beginner tilt te doen slaan bij de introductie van nieuwe concepten. Het is veel beter om concepten stap voor stap te kunnen uitleggen zonder ooit de mogelijkheid te moeten missen om voorbeelden in praktijk te kunnen tonen. Een gebruiker zal trouwens heel snel merken dat hij of zij vrijwel onmiddellijk met objecten leert omgaan, en de taal LPC is dusdanig opgebouwd dat alle ietwat complexere zaken echt veel makkelijker gaan met behulp van objecten.

De scheiding tussen klassieke datatypes en klassen is in LPC zeer strict. In feite kan je geen echte datatypes definiëren door middel van klassen. Alle objecten zijn van het (klassieke) type *object*. Dit is te verklaren met het feit dat elke klasse automatisch[15] een klasse *object* via inheritance opneemt in zichzelf. Alle klassen zijn dus te beschouwen als specialisaties van eenzelfde klasse, wat op basis van de standaard "is een" interpretatie van inheritance kan gelezen worden als "elke instantiatie is een object"[16]. Dit geeft tevens de verbinding tussen klassieke types en objecten en klassen.

Volgende datatypes zijn aanwezig in LPC:

int Het int type wordt gebruikt om gehele getallen voor te stellen. Het bereik van de waarden van dit type is $-2^{31} \dots 2^{31} - 1$.

string Het string type beslaat alle reeksen van lettertekens. Het kan vergeleken worden met packed arrays in Pascal, en char[] in C en C++. Speciaal aan deze strings[17] is dat de lengte dynamisch is. In vele talen moet je bij de definitie van een variabele de maximale lengte opgeven. Dit is niet nodig in LPC.

float Het float type was in het begin niet aanwezig in LPC. De DGD implementatie heeft dit type dusdanig gedefiniëerd dat een type conversie van int naar float, en terug geen verlies van nauwkeurigheid geeft. Het bereik van dit type is vastgelegd als $2.22507385851e^{-308} \dots 1.79769313485e^{308}$ met $\epsilon = 7.2759576142e^{-12}$.

object Het object type is zoals eerder vermeld de verbinding tussen klassieke datatypes en klassen en objecten. Elk object kan aan een variabele van dit type toegekend worden, en alle operaties die op dit type toepasbaar zijn, zijn dus toe te passen op eender welk object (onafhankelijk van klasse).

mapping Het mapping type is reeds eerder vermeld geworden. Het is in feite een associatief array waarin (sleutel, waarde)–paren worden bijgehouden. Zowel de sleutel als de waarde zijn van het type mixed. Dit wordt door een deel van de LPC–wereld beschouwd als een voordeel, en door de anderen als een fout. In feite zou het beter

[15] De gebruiker heeft hier helemaal geen controle over.

[16] Hier wordt de term instantiatie gebruikt om verwarring te voorkomen. Deze instantiaties worden algemeen doorheen de thesis objecten genoemd.

[17] Variabelen en constanten van dit type noemt men strings, en het lijkt duidelijker dan bijvoorbeeld "letterreeksen".

en mooier zijn indien het mogelijk was op te geven wat de types moeten zijn van sleutel en waarde. De syntax en semantiek van de taal laten dit echter niet toe.

mixed Het mixed type is een overblijfsel van de oudere LPmud versies waarin datatypes niet bestonden, of genegeerd werden. Om gelijkvormig te blijven met deze versies werd een type ingevoerd dat ale andere types overkoepelt. Een variabele van type mixed kan waarden bevatten uit eender welk ander type. Uiteraard is het gebruik van dit type enkel nuttig bij definitie van algemene functies zoals add(), en wordt het gebruik ervan ten zeerste afgeraden.

Omdat arrays veel voorkomen in programmas is hiervoor ook een constructie toegevoegd aan de taal[18]. Min of meer in analogie met C en C++ wordt gebruik gemaakt van het *–symbool om aan te geven dat een array van een gegeven type bedoeld wordt. Figuur 2.3 geeft een aantal voorbeelden. Zoals men kan zien is het mogelijk meerdimensionale arrays te definiëren door het gebruik van meerdere *–symbolen. Merk op dat een variabele van het mixed type ook arrays van willekeurig dimensie kan bevatten.

```
int *array_of_integers;
string **array_of_arrays_of_strings;
mapping *array_of_mappings;
mixed **array_of_arrays_of_anything;
```

Figuur 2.3: Voorbeelden van array definities

In tegenstelling tot andere talen is er geen automatische type conversie in LPC aanwezig. Een toekenning van een float aan een int is dus niet toegelaten. Om toch interactie tussen types mogelijk te maken, werd naar voorbeeld van C het concept van expliciete type conversie[19] ingevoerd in LPC. De syntax is net zoals in C:

(type)uitdrukking

Het resultaat van een expliciete type conversie is uiteraard het gegeven type. Niet alle types kunnen in eender welk ander type omgezet worden. Tabel 2.3 geeft al de mogelijke overgangen (expliciete type conversie naar het eigen type is niet vermeld).

Tenslotte is het ook mogelijk de zichtbaarheid (bereik of bruikbaarheid) van een variabele aan te geven. Dit heeft voornamelijk invloed op inheritance en op de mogelijkheid de toestand van een object naar een bestand te schrijven en terug in te lezen. Volgende specificaties zijn aanwezig:

static Deze specificatie geeft aan dat de variabele géén deel uitmaakt van de wegschrijfbare toestand van de klasse. Bij wegschrijven wordt zulk een variabele genegeerd, en bij inlezen van een toestandsbeschrijving zal ook niet aan de waarde geraakt worden. Dit

[18]Een verdere bespreking van datatypes en de wijze waarop ze werken is te vinden op pagina 23.

[19]In vakliteratuur wordt dit *casting* genoemd. De nederlandse term is equivalent.

Van	Naar	Opmerkingen
int	float string	 Decimale voorstelling.
float	int string	Afronding naar boven. Decimale voorstelling.
string	int float	 String moet beginnen met een getal, mogelijk met spaties ervoor.
object mapping		 Geen conversie mogelijk.
mixed		Enkel in vorige gevallen.

Tabel 2.3: Expliciete type conversie

kan van belang zijn, omdat voor efficiëntie toestandsvariabelen die waarde 0 hebben, niet worden weggeschreven. Bij inlezen worden dan alle variabelen die wegschrijfbaar zijn op 0 gezet alvorens de toestand uit het bestand in te lezen.

private Deze specificatie geeft aan dat de variabele enkel gekend zal zijn binnen deze klasse. Bij een verdere specialisatie van deze klasse (door middel van inheritance) zal de variabele niet gekend zijn. Enige manier om de waarde ervan te bekomen of te veranderen is via functies die in de klasse aanwezig zijn. Dit beschermt de variabele dus tegen directe manipulaties. Om problemen bij multiple inheritance te vermijden, impliceert "private" tevens "static".

2.2.2 Klassen en entiteiten

In LPmud wordt er een erg kunstmatig en eigenzinnig onderscheid gemaakt tussen objecten en klassen. Neem in eerste instantie aan dat er geen klonen worden gemaakt van klassen. Dat betekent dat elk bestand met broncode aanleiding zal geven tot één enkel object. In de LPmud versie van LPC hebben alle objecten een omgeving en een inhoud, en indien nu een object zichzelf naar een omgeving verplaatst tijdens de initialisatie die gebeurt bij creatie, dan is dat object geen klasse. Indien anderzijds zulk een verplaatsing niet gebeurt, dan heeft het object geen omgeving, en is het een klasse.

In praktijk kan men echter klonen maken van klassen. En dat maakt het geheel zeer complex, onoverzichtelijk, en hoogst onbetrouwbaar. Het is namelijk zo dat een kloon van een klasse gemaakt kan worden zonder dat de broncode als object geladen is. In dat geval wordt de broncode vertaald, en uitsluitend als programmacode opgenomen zonder er een object mee te associëren. Hiervan worden dan klonen gemaakt. Uiteraard kan er wel een object geassocieërd worden met deze programmacode, en dat wordt dan de blauwdruk genoemd van de klasse. Indien de blauwdruk een omgeving heeft, kan men geen klonen meer maken van de klasse. En dit geeft dadelijk aan waar alles misloopt. Het is mogelijk

dat eerst een aantal klonen worden gemaakt van een klasse, waarna de blauwdruk geladen wordt die zichzelf naar een omgeving verplaatst, en dan blijkt het *niet* meer mogelijk te zijn klonen te maken van die zelfde klasse! Dit is een erg storende anomalie.

In de standaard DGD implementatie van LPC is er geen enkel onderscheid tussen objecten en klassen. Wel zijn er nog steeds blauwdrukken en klonen. Dit heeft wel degelijk zin omdat de associatie tussen blauwdruk en programmacode toelaat om een nieuwe versie van de programmacode op te laden door de blauwdruk die ermee samenhoort te vernietigen (het vernietigen van een blauwdruk vernietigt ook de programmacode die ermee geassociëerd is[20]).

In sommige situaties kan het handig zijn toch objecten te maken die uniek zijn in de zin dat er geen klonen van gemaakt kunnen worden. Concreet betekent dit dat deze objecten niet als klasse kunnen optreden (indien het maken van klonen niet toegestaan is, is het toelaten van inheritance niet erg zinvol, temeer daar het object niet meer als klasse wordt beschouwd, en inheritance enkel op het niveau van klassen werkt). Men kan argumenteren dat het invoeren van een dergelijke faciliteit overbodig is. Kijk echter naar Eiffel waarin de klasse BOOLEAN slechts twee instantities heeft: *true* en *false*. Deze zijn uniek. Elke expressie die een waarheidswaarde geeft, heeft één van deze twee objecten als resultaat. Dus kan de gelijkheid van twee zulke resultaten makkelijk gecontroleerd worden door de gelijkheid van objecten te testen. Bij talen die de uniciteit van dergelijke objecten niet kunnen garanderen moet er getest worden op een waarde in de toestand van de objecten. Het blijkt dus een kwestie van keuze te zijn. Voor deze thesis werd gekozen om een constructie in te voeren in LPC die het mogelijk maakt unieke objecten te specifiëren, door te eisen dat in de broncode wordt aangegeven of het een klasse betreft of een entiteit[21] De syntax is voor beide specificaties gelijk (op de naam na, uiteraard):

class naam { broncode }
entity naam { broncode }

Van een klasse kunnen objecten bekomen worden door klonen. Het maken van een kloon van een entiteit is niet toegelaten, en zal dus een uitvoeringsfout tot gevolg hebben. Inheritance van een entiteit is eveneens niet toegelaten. In bovenstaande syntaxbeschrijving staat “naam” voor de naam van de klasse of entiteit (dit is een arbitraire string), en “broncode” voor de code die de toestand en het gedrag van de klasse of entiteit beschrijft.

2.2.3 Functieoproepen

Functieoproepen[22] zijn een zeer belangrijk deel van een programmeertaal. In object gerichte talen heb je meestal drie soorten functieoproepen (die dikwijls uniform uitgewerkt zijn).

[20] Uiteraard zullen alle klonen die afgeleid zijn van de “oude” programmacode blijven werken met die versie (dit geldt ook voor objecten die via inheritance afhangen van die code). De “nieuwe” code zal gebruikt worden voor de blauwdruk, nieuwe klonen, en objecten die via inheritance van de klasse afhangen en waarvoor de programmacode nog geladen moet worden.

[21] Unieke objecten worden in deze thesis entiteiten genoemd.

[22] In de vakliteratuur wordt dit dikwijls vermeld als “oproepen van methodes” of “invokaties van berichten”.

Na een kort overzicht zullen de drie soorten in detail besproken worden.

1 **intra–object oproepen**
Dit is het soort functieoproepen dat ook in klassieke talen te vinden is, soms met scheiding tussen procedures en functies zoals Pascal. LPC gebruikt de C–achtige functies en oproepen. Dit betekent voornamelijk dat zulk een scheiding niet bestaat. De oproepbare functies op een bepaald punt in het object zijn alle functies die op dat punt gedefiniëerd zijn[23]. Functiedefinities binnen in elkaar definiëren zoals dat mogelijk is in Pascal en Algol is niet mogelijk in LPC.

2 **intra–inheritance oproepen**
Deze soort functieoproepen is specifiek voor modulaire talen en object gerichte talen aangezien klassieke talen geen inheritance kennen. Het betreft oproepen van functies die binnen de inheritance structuur van de beschouwde klasse gedefiniëerd zijn. Strict genomen is de vorige groep oproepen een deelgroep van de intra–inheritance oproepen.

3 **inter–object oproepen**
Hier komt de echte object gerichtheid naar voren. Het betreft het oproepen van functies die in een ander object[24] aanwezig zijn. Deze groep functieoproepen is uiteraard een noodzaak om interactie tussen objecten mogelijk te maken.

Sommige talen zoals Self [US91] hebben alle functieoproepen uniform gemaakt, zodat je eigenlijk nog maar één type oproep hebt[25]. LPC beschouwt de drie soorten functieoproepen als syntactisch verschillende zaken, en dus zal dit onderscheid hier tot uiting komen. Het zal evenwel duidelijk worden dat de scheiding niet perfect is. Een simpel, doch afdoend voorbeeld zal alles duidelijker maken.

Intra–object functieoproepen

Figuur 2.4 geeft een mooi voorbeeld van een intra–object functieoproep. In objecten van klasse A doet de functie foo() een oproep naar bar(), binnen het object[26]. Vermits de functie bar() *na* functie foo() gedefiniëerd wordt, is een prototype noodzakelijk.

Een oproep van foo() met als argument 5, zal tot gevolg hebben dat bar() wordt opgeroepen met argument $5 + 1 = 6$, en functie bar() zal het getal 6 afdrukken op het scherm

[23] Dit door middel van een functiedefinitie of van een funtieprototype. Een functieprototype is een beschrijving hoe de functie opgeroepen moet worden, en laat toe dat de eigenlijke definitie verder in het object gebeurt.

[24] Hier moet men wel degelijk van verschillende objecten spreken, en niet klassen, omdat een functieoproep mogelijk is vanuit een object A naar een object B, met A en B beiden instantiaties van eenzelfde klasse.

[25] Self gaat nog verder door toestand ook via hetzelfde paradigma te benaderen. Een object in Self heeft toegang tot zijn toestand via berichten die het naar zichzelf stuurt.

[26] Bij intra–inheritance functieoproepen zal men zien dat de vermelding "objecten van deze klasse" hier cruciaal is. Indien de klasse A door inheritance aanwezig is in een andere klasse, kan de oproep van bar() intra–inheritance worden.

```
/*
 * This is class A.
 */
class A {
    void bar(int a);

    void foo(int b) {
        bar(b + 1);
    }

    void bar(int a) {
        write(a);
    }
}
```

```
/*
 * This is entity B.
 */
entity B {
    inherit "A";

    void bar(int a) {
        write(0);
    }
}
```

Figuur 2.4: Voorbeeld van functieoproepen: klasse A en object B

van de gebruiker. Figuur 2.5 geeft dit schematisch weer. Er wordt een speciale notatie gebruikt om aan te geven in welke klasse een bepaalde functie gedefiniëerd is:

$$\text{klasse::functie(argumenten)}$$

$$\text{A::foo}(5) \longrightarrow \text{A::bar}(5+1) \equiv \text{A::bar}(6) \longrightarrow \text{write}(6)$$

Figuur 2.5: Oproepvolgorde in klasse A

Dit is het klassieke concept van functieoproepen zoals die bijvoorbeeld in C bestaan.

Intra–inheritance functieoproepen

Indien inheritance gebruikt wordt, is het principe van functieoproepen iets ingewikkelder. Zoals men voor object B in figuur 2.4 kan zien, is er een herdefinitie mogelijk van functies. Zoals zal blijken kan dit met behulp van een aantal specificaties onder controle gehouden worden.

Om te begrijpen wat er gebeurt in figuur 2.4 wanneer er een functie wordt opgeroepen, geeft tabel 2.4 de functietabel van klasse A, en de functietabel van object B. Voor de intra–object functieoproepen in een object van klasse A wordt de linkertabel gebruikt (met bovenvermeld resultaat), terwijl voor object B de rechtertabel gebruikt wordt. Dit ten gevolge van de herdefinitie van functie bar().

Indien in object B de functie foo() wordt opgeroepen, zal een zoektocht binnen dde functietabel voor B leiden tot de ontdekking dat de definite van foo() in klasse A te vinden is. Deze functie zal uitgevoerd worden, en zoals reeds gezien is, wordt hierin een functie

Functie	Plaats
bar()	Klasse A
foo()	Klasse A

Functie	Plaats
bar()	Object B
foo()	Klasse A

Tabel 2.4: Functietabellen voor klasse A en object B

bar() opgeroepen. Wederom wordt de tabel met functie voor object B geraadpleegd, en de definitie van bar() blijkt in object B zelf te zitten, dus die wordt opgeroepen.

Net zoals voor de intra–object oproepen kunnen we hier nagaan hoe een oproep van de functie foo() met argument 5 in B verloopt. Zie hiervoor figuur 2.6.

$$\text{A::foo}(5) \longrightarrow \text{B::bar}(5+1) \equiv \text{B::bar}(0) \longrightarrow \text{write}(0)$$

Figuur 2.6: Oproepvolgorde in object B

Zoals reeds vermeld werd, kan het gedrag van een intra–inheritance oproep door middel van specificaties bij de functiedefinitie beïnvloed worden. Zo is het mogelijk aan te geven dat een functie niet *mag* geherdefiniëerd worden, of dat een functie onzichtbaar moet zijn in geval van inheritance. Volgende specificaties zijn voorradig:

nomask Deze specificatie geeft aan dat de functie niet mag geherdefiniëerd worden door klassen die verder in de inheritance structuur zitten. Dit geeft de garantie dat vanaf dit punt in de inheritance structuur alle oproepen van de functie naar deze definitie zullen wijzen. Het geeft dus een soort "cut" in de mogelijke herdefinities. Zelfs een "private" functie is na een "nomask" niet meer mogelijk.

private Deze specificatie geeft aan dat de functie enkel zichtbpaar moet zijn binnen de klasse waarin ze gedefiniëerd wordt. Dit betekent dat zelfs indien een functie met dezelfde naam gedefiniëerd wordt in een klasse die verder in de inheritance structuur zit, toch de locale functie zal worden opgeroepen.

In figuur 2.7 wordt een voorbeeld gegeven van het gebruik van deze specificaties. Hoewel het voorbeeld zeer kunstmatig is, geeft het toch een goed beeld van de mogelijkheden. Allereerst moet opgemerkt worden dat entiteit FNURT niet met succes vertaald zal kunnen worden, omdat de functie bar() geherdefiniëerd wordt, en in klasse APA (die via inheritance in BEPA zit, en dus zo wederom via inheritance in FNURT terecht komt) wordt gespecifiëerd dat bar() niet geherdefiniëerd mag worden. Indien we nu de definitie van bar() wegdenken in FNURT, dan zal de vertaling wel succesvol zijn. Wat gebeurt er nu bij een oproep van foo() in FNURT? In figuur 2.8 wordt de uitvoer gegeven, en hieruit is dadelijk af te leiden wat het effect van de "private" specificatie is.

Merk op dat in object FNURT een oproep "::bar(10)" vermeld wordt. Dit heeft een speciale betekenis (die verder nog uitgebreid zal worden). Indien namelijk de functie bar() in klasse APA niet nomask zou zijn, is een herdefinitie van bar() mogelijk. Nu kan het soms noodzakelijk zijn om toch nog toegang te hebben tot de originele functie bar(). Dit

```
/*
 * This is class APA.
 */
class APA {
    private void foo() {
        write("apa-foo\n");
    }

    nomask int bar(int i) {
        return i * 2;
    }

    void baz() {
        foo();
    }
}
```

```
/*
 * This is class BEPA.
 */
class BEPA {
    inherit "APA";


    private void foo() {
        write("bepa-foo\n");
    }

    void baz2() {
        foo();
    }
}
```

```
/*
 * This is entity FNURT.
 */
entity FNURT {
    inherit "BEPA";

    void foo() {
        write("Calling functions...\n")
        baz();
        baz2();
        write(::bar(10));
        write("Done.\n");
    }

    int bar(int i) {    /* this redefinition is an ERROR */
        return i + 1;
    }
}
```

Figuur 2.7: Gebruik van “nomask” en “private”

```
CALLING FUNCTIONS...
APA–FOO
BEPA–FOO
20
DONE.
```

Figuur 2.8: Uitvoer van FNURT::foo()

kan door middel van de "::" prefix bij een functieoproep. Het geeft aan dat de definitie van de op te roepen functie hoger[27] in de inheritance structuur moet gezocht worden. Het wordt meestal gebruikt waar een functie geherdefiniëerd moet worden om extra werk te doen, terwijl de originele defintie ook nog moet uitgevoerd worden. Zoals later zal blijken is de creatie functie hier meestal een extreem voorbeeld van.

Een laatste aspect van intra–inheritance functieoproepen is de zogenaamde *geetiketteerde inheritance*. Dit is eigenlijk een aspect dat bij het concept van inheritance hoort, doch vermits het hier zeker van belang is, wordt het reeds vermeld. Details betreffende deze vorm van inheritance zijn te vinden bij de bespreking van inheritance (pagina 28). Met een paar aanpassingen is het vorige voorbeeld hier ook te gebruiken. Figuur 2.9 geeft de aangepaste versies van de klassen en de entiteit FNURT. De eerder vermelde syntax *klasse::functie()* wordt hier in de taal gebruikt om aan te geven in welke klasse een bepaalde functie moet opgeroepen worden. De vermelding van het etiket dat bij de inherit–instructie staat, is nodig omdat de functie in meerdere klassen gedefiniëerd kan zijn[28]. De uitvoer van dit voorbeeld is gelijk aan wat er in het vorige voorbeeld verkregen werd (figuur 2.8).

Inter–object functieoproepen

Inter–object functieoproepen zijn uiteraard evenals de vorige soort specifiek voor object gerichte programmeertalen. In vele gevallen lijken functieoproepen tussen objecten erg op intra–object oproepen. Er wordt ook naar de functie gezocht in de functietabel van het ontvangende object. Een oproep zal dus ook steeds naar de *meest recente*[29] definitie van een functie gaan. In die zin volgt een inter–object functieoproep dezelfde semantiek als een intra–object functieoproep. Net zoals de intra–inheritance oproep een uitbreiding had op die semantiek, hebben ook oproepen tussen objecten een aantal nieuwe aspecten, eigen aan de soort.

Beschouw figuur 2.10. De functie baz() in object C geeft dadelijk de syntax van een inter–object oproep:

[27]Bij inheritance spreekt men dikwijls van hoger en lager. Dit moet gezien worden in functie van een boom (hoewel de inheritance structuur ook een grafe kan zijn). Het uiteindelijke object is dan de wortel, en klassen die geen inheritance gebruiken in hun definitie zijn de bladeren.

[28]Merk op dat als een functie de nomask specificatie heeft, een herdefinitie niet mogelijk is, zelfs niet met geëtiketteerde inheritance of iets dergelijks.

[29]Meest recent betekent hier dat de functiedefinitie de allerlaatste is van een reeks herdefinities van een gegeven functie.

```
/*
 * This is class APA.
 */
class APA {
    private void foo() {
        write("apa-foo\n");
    }

    void baz() {
        foo();
    }
}
```

```
/*
 * This is class BEPA.
 */
class BEPA {
    private void foo() {
        write("bepa-foo\n");
    }

    void baz() {
        foo();
    }
}
```

```
/*
 * This is entity FNURT.
 */
entity FNURT {
    inherit A "APA";
    inherit B "BEPA";

    void foo() {
        write("Calling functions...\n");
        A::baz();
        B::baz();
        write("20\nDone.\n");
    }
}
```

Figuur 2.9: FNURT met geëtikeerde inheritance

```
/*
 * This is entity (unique object) C.
 */
entity C {
    void baz() {
        B->foo(10);
    }
}
```

Figuur 2.10: Voorbeeld van functieoproepen: object C

object->functie(argumenten)[30].

Merk op dat hier wel degelijk een *object* opgegeven moet worden, en niet een klasse. De functieoproep is immers van object naar object. Het bekende verhaal is hier weer van toepassing. De oproep veroorzaakt een zoektocht in de functietabel van het object waarin een functie moet worden opgeroepen, en indien de functie gevonden wordt, kan de oproep uitgevoerd worden.

Dit is een zeer belangrijk aspect van inter–object functieoproepen. Het is mogelijk dat een object probeert een onbestaande functie op te roepen in een ander object. Vermits de vertaling van klassen volledig gestuurd wordt door de nood aan een bepaalde klasse ten gevolge van inheritance, inter–object oproepen en klonen, is het tijdens de vertaling van een klasse niet mogelijk te bepalen of een inter–object functieoproep naar een bestaande functie wijst[31]. De standaard DGD implementatie behandelt dergelijke functieoproepen erg dubieus. Zulk een functieoproep geeft namelijk gewoon de waarde 0 als resultaat. Binnen de wereld van MUDs is dit aanvaardbaar, en wordt het zelfs veelvuldig misbruikt. Voor meer serieuze doeleinden zoals deze thesis is een dergelijk gedrag echter onaanvaardbaar. Het LPC dialect dat hier gebruikt wordt zal dan ook een uitvoeringsfout genereren indien een onbestaande of “static” functie opgeroepen wordt via een inter–object functieoproep. Bij intra–object en intra–inheritance oproepen kan de controle reeds bij vertaling gebeuren, en dit is reeds het normale gedrag van DGD.

Een functie met een “static” specificatie heeft de eigenschap dat oproepen vanuit andere objecten niet toegelaten zijn. Functies die als “private” gespecifiëerd zijn, zijn dan ook automatisch static. Met betrekking tot inter–object oproepen heeft de nomask specificatie nog een handig neveneffect. Vermits herdefinitie van zo een functie niet toegelaten is, kan een willekeurig object vast steunen op het feit dat de nomask–versie van de functie opgeroepen zal worden. In MUDs wordt dit voornamelijk gebruikt voor veiligheid. Vele klassen zijn reeds voorgedefiniëerd, en hun samenwerking steunt dikwijls op de zekerheid dat een inter–object functieoproep het gewenste resultaat geeft. Nomask geeft de zekerheid dat een programmeur die bepaalde functie niet kan herdefiniëren.

Samenvatting

Volgende aspecten zijn belangrijk in verband met functieoproepen:

- Een **nomask** functie kan niet geherdefiniëerd worden.
- Een **private** functie kan enkel opgeroepen worden vanuit een functie die binnen dezelfde klasse gedefiniëerd is.

[30] Er zijn in feite twee manieren om een inter–object oproep in LPC te schrijven. De meest gebruikte methode is de -> operator. Indien echter een hogere graad van flexibiliteit gewenst is, kan gebruik gemaakt worden van de instructie call_other(object, functie, argumenten). Hierin is de functie een string die niet constant moet zijn. Het mag dus ook een variabele of een uitdrukking zijn.

[31] Indien de klasse waartoe het opgeroepen object behoort reeds geladen is, kan deze verificatie mogelijk wel gebeuren. De ontvanger van de inter–object oproep is echter niet noodzakelijk een constante en dan kan er geen beslissing genomen worden.

- Een **static** functie kan niet opgeroepen worden vanuit een ander object (een oproep met de `->` operator of call_other() vanuit een object naar een static functie in zichzelf is wel mogelijk.).
- Een oproep van een onbestaande of static functie geeft een uitvoeringsfout.

2.2.4 Parameterbindingen en datatypes

Het mechanisme dat gebruikt wordt om argumenten door te geven bij het oproepen van een functie wordt de parameterbinding genoemd[32]. Algemeen worden er drie mechanismen beschouwd[33]:

Call–by–value Bij dit mechanisme wordt de waarde van een argument onmiddellijk berekend en doorgegeven aan de opgeroepen functie. In die functie zal deze waarde in een lokale variabele opgeslagen worden, en aanpassingen ervan zijn lokaal voor de functie.

Call–by–reference In dit geval wordt het adres van het argument doorgegeven aan de opgeroepen functie. Via dit adres kan de waarde van het argument bekomen worden, en tevens veranderd worden. Met deze binding zullen lokale veranderingen zichbaar zijn in de oproepende functie[34].

Call–by–name Dit mechanisme geeft in feite een instructieadres door dat wijst naar de programmacode die de waarde van het argument berekent. Wanneer de parameter gebruikt wordt, zal die code opgeroepen worden om de juiste waarde te bekomen. In vele talen is dit mechanisme niet aanwezig.

Waarde–types	Wijzer–types
int	alle arrays
string	mapping
float	object

Tabel 2.5: Waarde–types en wijzer–types

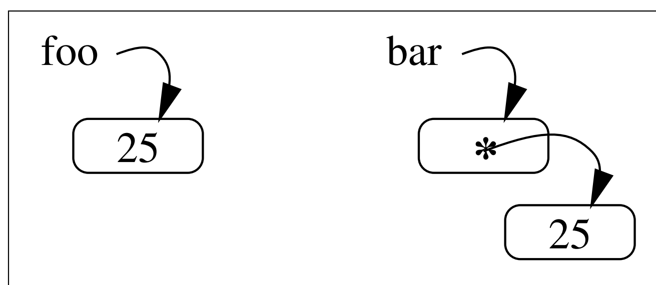
In LPC is er in feite slechts één van deze mechanismen geïmplementeerd, namelijk call–by–value. Het is echter zo dat de datatypes in deze taal in twee klassen opgesplitst kunnen worden. Enerzijds zijn er waarde–datatypes en anderzijds wijzer–datatypes. Tabel 2.5

[32]De bespreking van de parameterbinding mechanismen is gebaseerd op [DM93, pagina 21 e.v.].

[33]Engelse benamingen worden hier gebruikt omdat deze ingeburgerd zijn in de informaticawereld. Vertalingen zouden eerder tot verwarring leiden.

[34]Dit geeft problemen indien het argument van de oproep een expressie is in plaats van een variabele. Sommige talen zullen in dit geval een hulpvariabele invoeren terwijl andere talen in dit geval call–by–reference niet toelaten. LPC heeft geen call–by–reference bij de parameterbinding.

geeft de onderverdeling van de datatypes in deze twee groepen. Allereerst moet opgemerkt worden dat het datatype “mixed” niet in de tabel voorkomt. Dit is uiteraard een gevolg van het feit dat een variabele van dit type tot elk van de andere datatypes kan behoren, en dus zowel een waarde–type als een wijzer–type kan zijn. Verder is het “object” datatype moeilijk in één van de twee groepen onder te brengen omdat de bewerkingen die mogelijk zijn op dit type eigenlijk geen uitsluitsel geven over de aard ervan. Vermits de vertolker dit type als een wijzer–type behandelt, wordt het als wijzer–type opgenomen in tabel 2.5.



Figuur 2.11: Waarde–types en wijzer–types

Variabelen van een waarde–type gedragen zich zoals variabelen in de meeste klassieke talen. Alle bewerkingen werken op de waarde van de variabele, en in veel gevallen passen ze deze waarde ook aan. Doorgeven van deze variabele als argument naar een functie verloopt volledig volgens call–by–value. Bij een wijzer–type ligt de situatie iets anders. Variabelen van zulk een type hebben in plaats van een waarde een wijzer naar een waarde. In figuur 2.11 wordt het verschil grafisch voorgesteld. Zoals zal aangetoond worden, gebeurt het doorgeven van wijzer–variabelen ook volgens call–by–value, maar door de extra indirectie zal de waarde waar naartoe verwezen wordt zich gedragen volgens call–by–reference. Echte call–by–reference is echter niet mogelijk in LPC. In object gerichte talen is dit eerder een voordeel dan een nadeel, omdat het de scheiding en het verbergen van de toestand van objecten ten goede komt. Op basis van een vergelijking van het gedrag van de twee groepen datatypes zal geïllustreerd worden hoe de scheiding van objecten doorbroken wordt bij het gebruik van wijzer–types. Hieruit volgt een anomalie die in het volgende deel besproken wordt.

Figuur 2.12 geeft twee programmafragmenten die het gebruik van de twee groepen datatypes illustreren. Links wordt met het waarde–type “int” gewerkt en rechts met het wijzer–type “int array”.

- **Lijn 2:** De gebruikte variabelen moeten gedefiniëerd worden voordat ze gebruikt kunnen worden.
- **Lijn 4:** Aan de variabele “a” wordt een initiële waarde toegekend. Voor het waarde–type is dit een geheel getal, en voor het wijzer–type een array met als enige element datzelfde getal[35]. Figuur 2.11 geeft deze situatie grafisch weer.

[35]De notatie voor constanten van een array type is ietwat ongewoon. De elementen worden gescheiden door kommas, en het geheel wordt omsloten door accolades en haken. Voorbeeld: $(\{25, 365, -226\})$

```
1: /* waarde-type */    /* wijzer-type */
2: int a, b;            int *a, *b;
3:
4: a = 25;              a = ({ 25 });
5: b = a;               b = a;
6: b = 20;
7:                      b[0] = 20;
8: b = b + 8;           b = b + ({ 8 });
```

Figuur 2.12: Waarde–types en wijzer–types in LPC

- **Lijn 5:** De waarde van "a" wordt aan "b" toegekend. De inhoud van variabele "a" wordt gecopiëerd. Links betekent dit dat de waarde 25 gecopiëerd zal worden, terwijl rechts de *wijzer* naar het singleton ({25}) gecopiëerd wordt. Het verschil zal dadelijk duidelijk worden.
- **Lijn 6:** De variabele "b" in het voorbeeld van waarde–types krijgt een nieuwe waarde, namelijk het getal 20. In het geheugengebied waar de waarde van "b" opgeslagen is, komt nu het getal 20 te staan, terwijl het geheugengebied waar de waarde van "a" opgeslagen is, nog steeds 25 zal zijn. De variabelen "a" en "b" zijn niet meer gelijk.
- **Lijn 7:** In het voorbeeld van waarde–types wordt de variabele "b" nu ook aangepast, doch op een andere manier. De waarde van "b" wordt niet gewijzigd, maar de *inhoud van het array waar "b" naar verwijst* wordt gewijzigd. Vermits de variabele "a" naar hetzelfde array verwijst, zal de aanpassing ook hier merkbaar zijn. De twee variabelen zijn nog steeds gelijk (de inhoud van het geheugengebied dat hun waarde bevat is namelijk nog steeds hetzelfde, namelijk de wijzer naar het array).
- **Lijn 8:** Tenslotte wordt er een optelling uitgevoerd voor beide types. Voor het waarde–type is dit simpelweg een optelling van twee gehele getallen, en net als in lijn 6 zal nu de waarde van "b" vervangen worden door de nieuwe waarde 28. Bij het wijzer–type gebeurt eigenlijk hetzelfde. De optelling van twee arrays geeft een nieuw array dat de aaneenschakeling vormt van de gegeven arrays. Dit betekent concreet dat de waarde van "b" nu *wel* verandert! Een wijzer naar het nieuwe array wordt er namelijk in opgeslagen. Nu zullen "a" en "b" niet meer gelijk zijn.

Vermits alle datatypes via het call–by–value mechanisme worden doorgegeven aan functies, waarbij de waarde van een wijzer–type de wijzer is naar de effectieve waarde, zullen aan de lokale variabelen die de argumenten van de opgeroepen functie bepalen al deze waarden toegekend worden. Voor "int" en "float" is het gedrag van deze lokale variabelen voor de hand liggend. Bij strings is het iets complexer omdat er bijvoorbeeld één letter in een string aangepast kan worden. Het gedrag van waarde–types blijkt echter consistent te zijn, doordat zulk een aanpassing gewoon een nieuwe string als resultaat heeft.

```
void foo() {
    int *a;

    a = ({ 1, 2, 3 });
    bar(a);
}
```

```
void bar(int *arg) {
    arg[2] = 4;
}
```

Figuur 2.13: Parameterbinding met wijzer–types

Wijzer–types gedragen zich bij toekenningen aan elementen[36] anders, doordat er een wijziging gebeurt in het deel van het array of de mapping waar naartoe verwezen wordt (de waarde). Wijzigingen aan de inhoud van een wijzer–type zijn dus merkbaar in de oproepende functie. Neem figuur 2.13 ter illustratie. Na de oproep van de functie bar() zal de lokale array variabele "a" in foo() de waarde $(\{1, 2, 4\})$ hebben. Indien een functie in een ander object wordt opgeroepen, kan eenzelfde verandering aan het array waarnaar "a" wijst gebeuren. Dit doorbreekt dus de scheiding van de objecten omdat een *vreemd* object de toestand van het oproepende object aan kan passen via functieparameters met wijzer–types. In een meer pure visie van object gerichtheid zouden dergelijke aanpassingen enkel kunnen gebeuren door middel van functieoproepen in het object in wiens toestand het array is vervat. Dit zou betekenen dat het gedrag van wijzer–types afhankelijk is van het soort functieoproep waarbij het als argument wordt meegegeven. Voor intra–object en intra–inheritance oproepen is het gewenst een call–by–reference gedrag te hebben, terwijl voor inter–object oproepen dit gedrag ongewenst zou zijn. Deze dualiteit is ook verre van ideaal, en het lijkt dan ook best het normale LPC gedrag te houden.

```
entity "OSTKAKA" {
    int *arr;

    void src() {
        arr = ({ 1, 2 })
        "OL"->dst(arr);
    }


}
```

```
entity "OL" {
    int *ay;

    void dst(int *a) {
        ay = a;
    }
    void chg() {
        ay[0] = 0;
    }
}
```

Figuur 2.14: De wijzer–type anomalie

[36]Deze meeste wijzer–types zijn arrays en mappings. Voor het object datatype zijn er zo geen bewerkingen die invloed hebben op de inhoud, en dus is de veralgemening te verantwoorden.

2.2.5 De wijzer–type anomalie

Uit bovenstaande bespreking is makkelijk af te leiden dat wijzer–types nog een belangrijk neveneffect hebben. Bekijk de code in figuur 2.14. Na de uitvoering van de functie src() hebben de beide entiteiten een verwijzing naar hetzelfde array. Indien in een later stadium de functie chg() opgeroepen wordt in entiteit OL zal het array aangepast worden, en deze aanpassing zal zichtbaar zijn in de toestand van beide entiteiten. Dit is een verder gevolg van het call–by–reference gedrag van wijzer–types.

Het gedrag is op zich reeds onaangenaam omdat het toelaat dat objecten volledig naar willekeur de toestand van andere objecten kunnen aanpassen. Bij een object gericht paradigma is het belangrijk niet te moeten vertrouwen op de betrouwbaarheid van andere objecten. De situatie is echter nog erger doordat er een *swapping*[37] mechanisme in DGD aanwezig is. Dit mechanisme zorgt ervoor dat objecten die een tijd niet gebruikt worden naar een secundair geheugen geschreven worden om het primair geheugen vrij te maken voor de aktieve objecten. Omdat wijzers niet weggeschreven kunnen worden, heeft Felix Croes verkozen om variabelen van objecten die tot een wijzer–type behoren geheel weg te schrijven. Dit betekent concreet dat de inhoud van die variabelen tijdens het wegschrijven gecopiëerd wordt. Bij het terug inlezen zullen deze toestandsvariabelen dus een wijzer hebben naar een eigen exemplaar van de waarde in plaats van een wijzer naar een gedeeld exemplaar. Voor de entiteiten in figuur 2.14 betekent dit dat zodra één van beide entiteiten wordt weggeschreven wegens swapping, het array “arr” in OSTKAKA en het array “ay” in OL niet meer gelijk zullen zijn[38]

De anomalie zit nu in het feit dat men geen enkele controle heeft over het moment waarop swapping van een object plaats vindt. Zodoende heeft men dus geen controle over het moment waarop variabelen die tot een wijzer–type behoren een eigen exemplaar van hun effectieve waarde krijgen. Gedurende een tijd kan er dus een gelijkheid van arrays bestaan die, zodra één van de participerende objecten door swapping weggeschreven wordt, verandert in een ongelijkheid. En dit zonder dat de programmeurs hier enige controle over hebben!

Verschillende oplossingen dienen zich aan, doch elk met hun eigen specifieke problemen:

- Allereerst kan er gekozen worden om het call–by–reference gedrag van wijzer–types op te geven. Zoals in het vorige deel aangegeven werd, is dit niet aanvaardbaar als oplossing vermits het aanleiding geeft tot een dualiteit in de werking van wijzer–types.
- Een andere mogelijke oplossing is het verbieden van de toekenning van variabelen van een wijzer–type aan globale variabelen in objecten. Dit is duidelijk een onaanvaardbare manier om deze anomalie te behandelen.
- Ook is het mogelijk om bij toekenning tussen een doorgegeven variabele en een globale variabele in een object automatisch een lokaal exemplaar aan te maken van de

[37] Deze term uit de wereld van besturingssystemen heeft geen goed nederlands equivalent.

[38] De inhoud van beide arrays zal nog wel hetzelfde zijn, natuurlijk. Herinner dat de gelijkheid van arrays in feite de gelijkheid van de wijzers betreft.

effectieve waarde. Dus voor wijzer–type variabelen wordt dan een copie gemaakt van de elementenverzameling. Dit leidt weer tot een dualiteit, dit keer bij de werking van de toekenning.

- Verder kan men verkiezen om met behulp van truukjes toch wijzers weg te schrijven[39]. Dit laat echter toe dat objecten echt representaties van wijzer–types delen en dit is niet object gericht.

- Tenslotte is het ook mogelijk een tussenweg te volgen. In plaats van een verbod op te leggen op bepaalde operaties, wordt het onvoorspelbare gedrag van het copiëren van elementenverzamelingen aangepakt. Het is beter lokale copieën te forceren op een voorspelbaar moment, dan aan de willekeur van het swapping mechanisme overgeleverd te zijn. De eerst voorgestelde oplossing is hier een sterk doorgedreven geval van, waarbij reeds werd vermeld dat het niet goed bevonden wordt. Het is beter om de copiëeroperatie uit te stellen totdat een uitvoeringscyclus ten einde is.

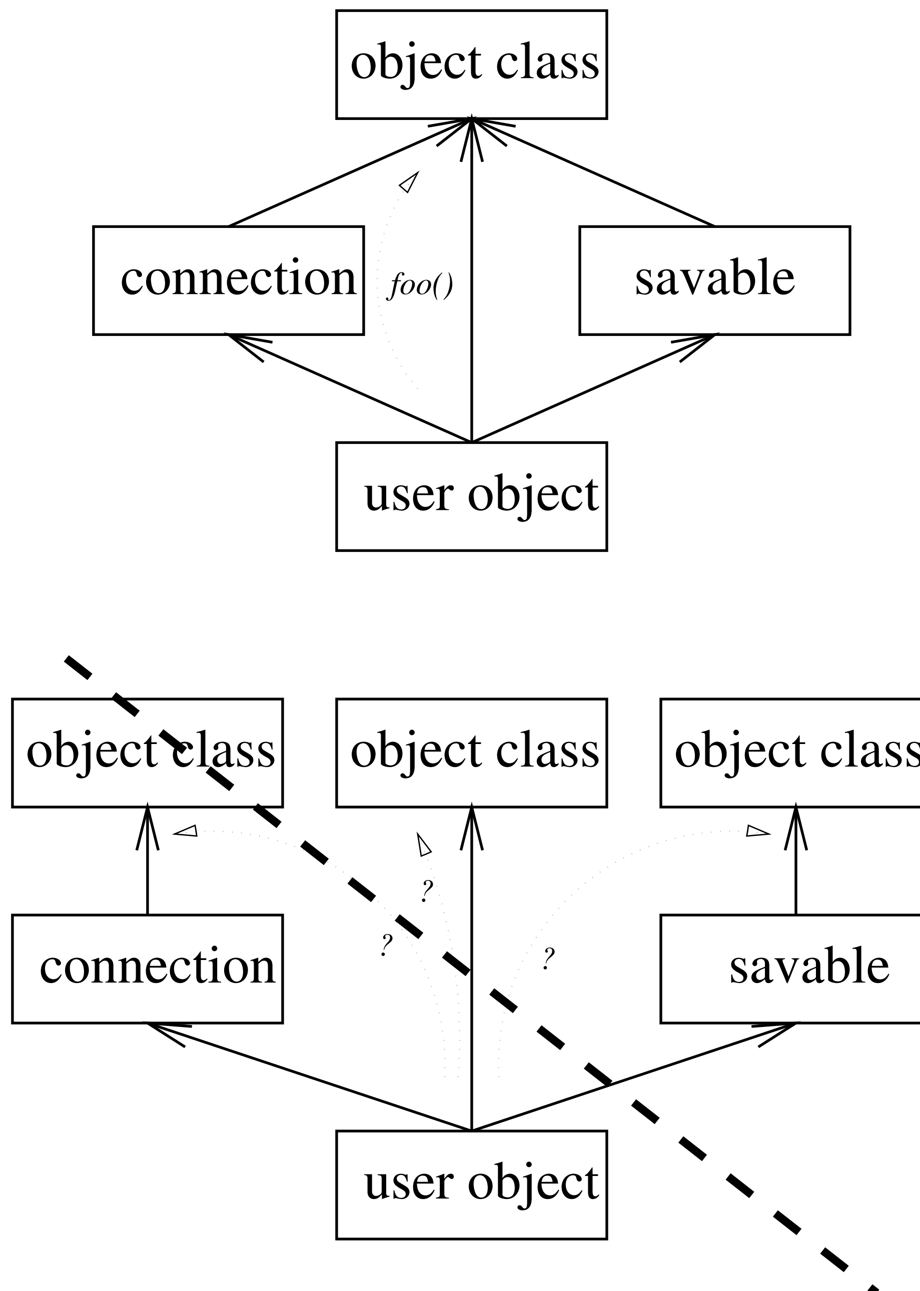
Zoals op pagina 36 zal worden besproken bevat DGD een strict sequentiële vertolker, en is alle uitvoering impulsgestuurd. De uitvoering van één impuls (of commando) moet dus beëindigd zijn alvorens een tweede kan aanvangen. In bovenstaande opsomming komt duidelijk tot uiting dat enkel de laatste oplossing aanvaardbaar is. Ook blijkt deze oplossing zeer goed te integreren in DGD. Tussen de uitvoeringscycli wordt er een stap toegevoegd die ervoor zal zorgen dat alle opgeslagen wijzers op het einde van die stap naar elementenverzamelingen wijzen die lokaal per object zijn. De wijzers die in de toestand van verschillende objecten opgeslagen zijn mogen na deze unificatiestap niet meer gelijk zijn. Op deze manier wordt de wijzer–type anomalie opgelost zonder dat er lastige beperkingen of dualiteiten tevoorschijn komen. Men kan argumenteren dat deze oplossing een beperking invoert, namelijk dat bijvoorbeeld arrays niet meer gedeeld kunnen worden tussen objecten. Deze mogelijkheid was echter reeds onzuiver uit object gericht oogpunt, en zoals hier beschreven werd, leidt het tot een anomalie ten gevolge van de swapping.

2.2.6 Inheritance

Inheritance is één van de basisconcepten van object gerichtheid, en werd al eerder vermeld in dit hoofdstuk (voornamelijk op pagina 17 e.v.). De bespreking hier zal dan ook redelijk bondig blijven.

LPC implementeert zogenaamde multiple inheritance, wat betekent dat een klasse (of entiteit) een specialisatie van meerdere klassen kan zijn, die op zich weer specialisaties kunnen zijn van meerdere klassen, enz. Ook is er geen beperking betreffende de inheritancestructuur. Eénzelfde klasse mag meer dan eens in de inheritancestructuur van een andere klasse voorkomen, omdat deze structuur een grafe vormt. Dit garandeert dat elke klasse in de inheritancestructuur slechts één keer in de specialisatieklasse wordt opgenomen.

[39]Dit kan men doen door de elementenverzamelingen apart weg te schrijven en dan in plaats van een wijzer, een index in de lijst verzamelingen weg te schrijven.



Figuur 2.15: Inheritancestructuren

Dit is nodig omdat er anders inconsistenties zouden kunnen ontstaan als gevolg van de verschillende paden die mogelijk leiden tot een bepaalde klasse in de inheritancestructuur. Vergelijk bijvoorbeeld de structuren in figuur 2.15 (de onderste structuur is ongeldig omdat er geen éénduidigheid bestaat betreffende in welke versie van de object klasse functie foo() moet gevonden worden).

Net zoals in vele andere object gerichte talen worden functies geherdefiniëerd wanneer een specialisatie van een klasse gedaan wordt[40]. Deze herdefinitie is globaal voor de gehele inheritancestructuur zodat oproepen van een functie steeds via een globale functietabel worden uitgewerkt. Uiteraard zijn er mechanismen voorradig om deze werkwijze toch nog te doorbreken op een gecontroleerde manier. Dit werd reeds besproken bij intra–inheritance functieoproepen op pagina 17. Toegang tot de variabelen in klassen in de inheritancestructuur kan beperkt worden door middel van specificaties zoals vermeld werd op pagina 13.

Het is belangrijk in te zien dat de achterliggende structuur van een klasse niet zichtbaar verondersteld dient te worden voor andere objecten. Het is in feite wel mogelijk om op te zoeken in welke klasse in de inheritancestructuur een bepaalde functie opgeroepen zou worden bij inter–object oproepen, en ook is er een primitieve functie om de lijst van klassen te bekomen die in de gegeven klasse via inheritance opgenomen zijn[41]. Voor normale toepassingen hoort hier echter nooit gebruik van gemaakt te worden.

2.2.7 Controlestructuren

Bij dit deel wordt enige kennis van klassieke programmeertalen verondersteld. De controlestructuren die in LPC aanwezig zijn, zijn namelijk volledig op C gebaseerd, en komen in werking volledig overeen met deze taal. Onder de naam controlestructuren nemen we ook contructies zoals de klasse/entiteit specificatie, de specificatie voor inheritance, . . .

```
class "class name" {          entity "entity name" {
    ...                           ...
}                             }
```

Figuur 2.16: Klassen en entiteiten (syntax)

Klassen en entiteiten

De specificatie van klassen en entiteiten werd reeds eerder besproken in dit hoofdstuk op pagina 14. Figuur 2.16 geeft nog een overzicht van de syntax van deze specificaties. Voor

[40] C++ is hier een tegenvoorbeeld, omdat er expliciet opgegeven moet worden dat een dergelijke herdefinitie gewenst is, namelijk met behulp van de “virtual” specificatie.

[41] Enkel het eerste niveau van inheritance kan bekomen worden. Dit volstaat omdat voor elk van die klassen wederom een lijst kan opgevraagd worden, en zo kan recursief de hele structuur opgevraagd worden.

details wordt verwezen naar de vroegere bespreking.

inherit "class name";
inherit labelname "class name";

Figuur 2.17: Inheritance (syntax)

Inheritance

Inheritance werd reeds uitgebreid besproken in dit hoofdstuk en hier zal dan ook enkel de syntax aangegeven worden met figuur 2.17. Zoals men kan zien zijn er twee vormen mogelijk. Allereerst kan een klasse gewoon voor inheritance opgegeven worden. Dit kan tot problemen leiden indien multiple inheritance in de klasse gebruikt wordt. In dat geval is het namelijk mogelijk dat functies in meerdere klassen dezelfde naam hebben, en dit maakt het onmogelijk de ene van de andere te onderscheiden. Om dit probleem op te lossen kan de tweede variant gebruikt worden. Hier kan een etiket geassociëerd worden met een klasse in de inheritancereeks, en dit etiket wordt dan gebruikt bij de functieoproep om aan te geven in welke klasse de functie gezocht moet worden.

Als een korte kanttekening is het misschien goed te vermelden dat er uiteraard ook een naamconflict kan ontstaan met de namen van variabelen. Dit conflict kan niet opgelost worden met etiketten omdat er geen constructie in LPC aanwezig is om een etiket op te geven bij een variabele. Enerzijds kan dit als een nadeel aangevoeld worden, doch anderzijds is het rechtstreeks gebruik van variabelen die in een generaliserende klasse te vinden zijn niet erg object gericht. De keuze die in LPC genomen werd, wordt niet ideaal beschouwd doch kan zonder enig probleem verdedigd worden. Net als bij vele andere problemen is er geen ideale oplossing voor alle talen te vinden.

Instructieblokken

Een instructieblok is een reeks van instructies, omsloten door accolades. Dit blok heeft dan weer de functie van een instructie. In een blok kunnen lokale variabelen gedefiniëerd worden, die ophouden te bestaan op het einde van het blok (dit mag triviaal lijken, doch in vroegere versies van LPC was dit niet het geval). Het voornaamste gebruik van instructieblokken komt tot uiting bij de definitie van functies waar de uitvoerbare code een instructieblok is, en bij de voorwaardelijke uitdrukkingen die verder besproken zullen worden, waar de instructie door middel van een blok uit meerdere instructies kan bestaan.

Functies

Functies vormen het voornaamste element in LPC, omdat alle bewerkingen op objecten via dit mechanisme verlopen. Figuur 2.18 geeft een voorbeeld van een functiedefinitie. Het betreft hier een functie die niet zichtbaar is voor andere objecten ten gevolge van de "static" specificatie (voor andere specificaties, zie pagina 18). Verder heeft de functie een

```
static string function(int arg) {
    int val;
    string desc;

    arg = random(arg);
    val = random(100);
    if (arg + val > 30)
        desc = "A very first description.\n";
    else
        desc = "This is the second description.\n";

    return desc;
}
```

Figuur 2.18: Function definition example

resultaatwaarde die van datatype “string” moet zijn. Indien er een “return” instructie in de functie is die een ander type[42] als resultaat teruggeeft, zal een vertaalfout aangegeven worden.

De functie heeft één parameter, namelijk een geheel getal dat aan de lokale variabele “arg” zal toegekend worden. Verder zijn er nog twee andere lokale variabelen, een geheel getal en een string, die beiden op 0 geïnitialiseerd zullen worden bij oproep van de functie. De werking van de instructies in deze functie is hier niet van belang. Wel moet opgemerkt worden dat de reeks van uitvoerbare instructies in de functie een instructieblok vormen. Een functie bestaat dus in feite uit een functiedefinitie en een instructieblok.

```
if (expr)
    statement1;
else
    statement2;
```

```
if (n == n1)
    statement1;
else if (n == n2)
    statement2;
else if (n >=n3 && n <= n4)
    statement3;
else
    statement4;
```

Figuur 2.19: Voorwaardelijke uitdrukkingen (syntax)

[42]Merk op dat de waarde 0 tot alle types behoort. Indien het type wordt opgevraagd van een variabele die 0 als waarde heeft, wordt echter T_INT gegeven, wat aangeeft dat het type “int” is.

```
switch(n) {
    case n1:     statement1;
                 break;
    case n2:     statement2;
                 break;
    case n3..n4: statement3;
                 break;
    default:     statement4;
}
```

Figuur 2.20: Switch instructie (syntax)

Voorwaardelijke uitdrukkingen

De *"if–else"* instructie wordt gebruikt om op basis van de waarde van een uitdrukking één of ander instructieblok uit te voeren. Het meest voorkomende gebruik hiervan wordt geïllustreerd in figuur 2.19. In het linkse stuk code wordt *statement1* uitgevoerd indien *expr* een niet–nul waarde heeft; in het andere geval wordt *statement2* uitgevoerd. De code die rechts staat, kan vervangen worden door een *switch* instructie zoals in figuur 2.20. In tegenstelling tot een aantal talen zoals C is het in LPC wel mogelijk om een waardeninterval op te geven bij een *case* label, door de eerste waarde en de laatste waarde gescheiden door twee puntjes op te geven.

```
do { statement; } while (expr);

while (expr) statement;

for (expr1; expr2; expr3) statement;
```

Figuur 2.21: Repeterende uitdrukkingen (syntax)

Repeterende uitdrukkingen

In LPC zijn er (net zoals in C en Pascal) drie verschillende manieren om een repeterende uitdrukking te schrijven. Figuur 2.21 geeft de mogelijke uitdrukkingen. De eerste versie betekent dat *statement* uitgevoerd zal worden totdat *expr* evalueerd als nul. De *repeat–until* uitdrukking in Pascal heeft dezelfde betekenis.

In de tweede vorm zal zolang *expr* evalueert als een niet–nul waarde, de instructie *statement* uitgevoerd worden. Dit komt overeen met de gelijknamige *while* uitdrukking in Pascal.

Tenslotte is er nog de meest algemene vorm, die eigenlijk enkel in C–achtige talen bestaat. Hier wordt allereerst *expr1* uitgevoerd. Daarna zal zolang *expr2* niet–nul evalueert, de instructie (of het instructieblok) *statement* uitgevoerd worden. Voor elke volgende evaluatie van *expr2* zal *expr3* uitgevoerd worden.

Tijdens de uitvoering van de te repeteren instructie (*statement*) kan men gebruik maken het *break* commando om de repeterende uitvoering af te breken. Met het *continue* commando kan op eender welke plaats in het instructieblok onmiddellijk naar het begin van de repeterende opdracht gegaan worden (de repetitie–voorwaarde wordt onmiddellijk geëvalueerd).

2.2.8 Waarom werden LPC en DGD gekozen?

De kritische lezer zal misschien opmerken dat een taal zoals C++ hier ook wel zou kunnen voldoen. Dit is deels waar. Bijna elke object gerichte taal is hier bruikbaar, voor wat betreft de programma–technische aspecten. Het is echter zo dat DGD meer biedt dan gewoon een programmeertaal. Zoals uit dit hoofdstuk blijkt, heeft LPC als programmeertaal zeker voldoende ondersteuning om object gericht programmeren aan te bieden. Zeker voor educatieve toepassingen is er een goede ondersteuning, indien men de doelstellingen volgt. Er is mogelijkheid te werken met klassieke principes (hoewel dit voor ietwat complexe opgaven zeker niet makkelijk is), en men wordt op een vrij intuïtieve manier in de richting van object gericht programmeren gedreven.

De voornaamste troef van de DGD/LPC combinatie is echter de opzet van de werkomgeving. Talen zoals Smalltalk–80 en Eiffel hebben wel een omgeving waarin de programmeur kan werken, maar het blijft beperkt tot een enkele gebruiker. Gebruik maken van elkaars werk kan enkel met het doorgeven van bestanden, al dan niet door middel van copiëren. Dit is echter niet goed in een object gerichte omgeving waar men zou verwachten dat er ook op dit niveau gebruik gemaakt zou worden van het herbruikbaarheidsprincipe.

DGD biedt de gebruikers een open, dynamische werkomgeving, die toegespitst is op samenwerking en een intuïtief en experimenteel leerproces. Samenwerking is zeker een pluspunt omdat het toelaat dat mensen elkaar nieuwe dingen leren. Ook draagt het bij tot het principe van de herbruikbaarheid van code. Klassen die door de ene gebruiker geschreven werden, kunnen mogelijk door een andere gebruiker toegepast worden in een ander project. Dit kan in DGD en LPC zonder dat er copieën van bestanden gemaakt moeten worden, en zonder dat er gewerkt moet worden met toegangspermissies. DGD biedt een echte abstracte wereld waarin gewerkt wordt. Dit betekent dat gebruikers niet in een eigen, geïsoleerde wereld zitten, maar eerder dat al wat gemaakt wordt, globaal aanwezig is en dan ook beschikbaar is voor iedereen.

LPC als taal heeft niet te kampen met lastige beperkingen zoals single inheritance (deze berperking is bijvoorbeeld aanwezig in Smalltalk–80), en heeft gelukkig niet de complexiteit en onzuiverheid die in C++ te zien is. Dit betekent niet dat die talen niet goed zijn, doch geeft wel aan dat binnen de doelstellingen dergelijke talen niet voldoen. Het is zeer waarschijnlijk dat nog andere talen gekozen hadden kunnen worden, zeker nu er een overvloed aan talen bestaat waarin object gericht geprogrammeerd kan worden.

Hoofdstuk 3

Implementatie

"It is a pity the universe doesn't use a segmented architecture with a protected mode."
(William Irving Zumwalt, in "Wizard's Bane")

De implementatie[1] van de thesis bestaat enerzijds uit de aanpassingen aan de LPC taal, en de daarmee verweven aanpassingen aan DGD, en anderzijds uit de klassen die de werkomgeving implementeren en klassen die voor algemeen gebruik beschikbaar worden gesteld. Zodoende wordt er dus in twee programmeertalen gewerkt: C en LPC. De voornaamste elementen die in dit hoofdstuk besproken zullen worden, zijn eerder van praktische aard. De redenen voor bepaalde aanpassingen aan DGD werden reeds besproken in het vorige hoofdstuk, en er zal dan ook steeds naar verwezen worden.

Het verloop van de implementatie is gedreven door de wens om steeds in staat te zijn het volledige geheel te kunnen testen. Aangezien DGD voornamelijk voor MUDs wordt gebruikt, werd hier de ideale test gevonden: gedurende de hele implementatie van aanpassingen aan DGD moet (met enig herwerken van de LPC code ten gevolge van taalveranderingen) de MUD omgeving zonder problemen blijven werken. Op deze wijze is een snelle controle mogelijk, en is er steeds een werkomgeving aanwezig. Door de flexibele opzet van LPC en DGD is het dan ook mogelijk nieuwe klassen in te voeren met behulp van de MUD–omgeving, om dan stelselmatig over te gaan op de werkomgeving die moet bekomen worden.

De aanpassingen aan DGD worden dusdanig uitgevoerd dat de nieuwe programmacode vrijwel geheel samenvloeit met de bestaande code. Dit geeft enerzijds een ordelijk programma (op voorwaarde dat de bestaande code ordelijk geschreven is, wat hier zeker het geval is), en anderzijds geeft dit aanleiding tot een meer doordachte aanpak omdat een conceptuele verandering moet passen in de concepten die reeds aanwezig zijn. DGD blijkt dit zeer goed toe te laten, ten gevolge van de heel modulaire structuur.

[1] Eigenlijk zouden hoofdstukken 3 en 4 "Hoe" en "Waarom" moeten heten, doch deze nederlandse uitdrukking geldt niet in de informatica (en in feite eigenlijk nergens) omdat je moeilijk over het "hoe" kan beslissen als je het "waarom" nog niet bepaald hebt. De inversie is echter wel waar.

3.1 Inleiding betreffende de werking van DGD

Alvorens de effectieve implementatie aan te pakken van de gestelde aanpassingen is een korte bespreking nodig over de werking van DGD. Dit zal voldoende inzicht geven in de materie om de draagwijdte van de aanpassingen aan DGD zoals deze in dit hoofdstuk beschreven worden juist in te schatten.

DGD is een netwerk server, en dit zal reeds bij een hoog–niveau beschrijving blijken. Ook zal het principe van de sequentiële uitvoering, en impuls–gestuurdheid dadelijk naar voren komen. De werking van DGD is op het hoogste abstractieniveau als volgt te beschrijven:

1 Initialiseer DGD.

2 Verander geïmporteerde arrays in eigen copieën.

3 Voer het volgende uit totdat "stop[2]" waar is:

- 3.1 Verwerk tijdgestuurde functieoproepen.
- 3.2 Indien "stop" niet waar is:
  - 3.2.1 Indien "data" kan gelezen worden van een netwerkverbinding:
    - 3.2.1.1 Roep *receive_message(data)* op in het user–object dat met deze connectie geassociëerd is.
    - 3.2.1.2 Verander geïmporteerde arrays in eigen copieën.
- 3.3 Kuis vernietigde objecten op.
- 3.4 Behandel geheugen, swap en dump[3].

4 Kuis alle datastructuren op en beëindig het programma.

In dit schema is duidelijk te zien dat DGD in feite in een (bijna) eindeloze lus werkt, waarbij in elke iteratie tijdgestuurde acties plaatsvinden en er ook telkens invoer van één gebruiker verwerkt wordt. Het zijn deze twee elementen die aanleiding geven tot uitvoering van LPC code, en zodoende dus de interactie verzorgen met de LPC omgeving.

Indien punt 31 verder bestudeerd wordt, blijkt dadelijk dat hier enkel intra–inheritance functieoproepen mogelijk zijn, en dan nog wel enkel ongeëtiketteerde oproepen. Ook moet opgemerkt worden dat een object enkel tijdgestuurde oproepen kan aanvragen naar functies in zichzelf. Dit volgt rechtstreeks uit de syntax van de kfun[4] *call_out()*:

---

[2]De stopconditie voor DGD wordt waar indien de variabele *stop* waar is. Deze variabele kan enkel deze waarde krijgen indien de kfun *shutdown()* uitgevoerd wordt.

[3]Hier is moeilijk een goede nederlandstalige term voor te vinden. Bij een "dump" wordt de gehele LPC omgeving naar een bestand geschreven. Dit bestand kan dan later gebruikt worden om de omgeving terug in te laden.

[4]Functies die in DGD zelf aanwezig zijn, noemt men kernfuncties (*kernel functions*) en dit wordt meestal afgekort tot "kfun".

call_out(functienaam, vertraging, argumenten...)

Dit betekent enerzijds dat een tijdgestuurde oproep nooit rechtstreeks aanleiding kan geven tot het vertalen van een klasse of het laden van een object. Anderzijds zijn de argumenten die in een tijdgestuurde oproep opgegeven worden altijd lokaal ten opzichte van het object zelf. Ze kunnen mogelijk geïmporteerd zijn uit andere objecten door functieoproepen die de oproep van de kfun *call_out()* voorafgingen maar zoals later beschreven zal worden, zijn het altijd waarden die in het gegevensblok van het object opgeslagen zijn.

Punt 3(2)12 betreft de tweede mogelijkheid tot interactie tussen de server en de LPC omgeving. Er wordt namelijk invoer gelezen van netwerkverbindingen, en deze invoer string (een reeks karakters afgesloten met een scheidingskarakter) wordt als argument meegegeven aan een functieoproep naar *receive_message()* in het object dat met die verbinding geassocieërd wordt. Hieruit blijkt dadelijk de opzet van DGD. Alle interactie tussen gebruikers en de LPC omgeving wordt in LPC objecten uitgevoerd. Dit geeft een grotere flexibiliteit, en is tevens ook een vervollediging van het principe dat zoveel mogelijk functionaliteit in LPC zelf uitgevoerd zou moeten worden.

Verder dient opgemerkt te worden dat punt 2 in de standaard implementatie van DGD niet bestaat. Het is een aanpassing die in het kader van deze thesis werd uitgevoerd. Verder in dit hoofdstuk zal daar verder op ingegaan worden als één van de grootste aanpassingen die aan DGD uitgevoerd zijn.

3.1.1 Geheugenbeheer, swap en dump

Er wordt in bovenstaand schema melding gemaakt van geheugenbeheer, swapping, en een dump. Het zijn net deze drie aspecten die enerzijds de mogelijkheid bieden om voor lange tijd (in theorie oneindig lang) zonder onderbreking te kunnen werken met DGD. In principe is het mogelijk om slechts één maal de server op te starten. Dit is mogelijk omdat het geheugenbeheer met een opkuisfunctie werkt (veel object gerichte talen hebben een dergelijke functie[5] Die zorgt ervoor dat het geheugen dat door vernietigde objecten wordt ingenomen na een tijdje weer vrij komt voor nieuwe objecten. Ook worden alle tijdelijke gegevens op deze manier vrijgegeven zodra er geen gebruik meer van gemaakt wordt. Dit gebeurt op een vrij complexe manier.

Allereerst moet opgemerkt worden dat het geheugen dat door DGD wordt gebruikt opgesplitst wordt in twee stukken:

statisch geheugen Het statisch geheugen wordt gebruikt om vaste gegevens in op te slaan. Dit betreft voornamelijk tabellen met een vaste grootte, toestandsinformatie die ten allen tijde aanwezig moet zijn (en dus niet onderhevig is aan swapping).

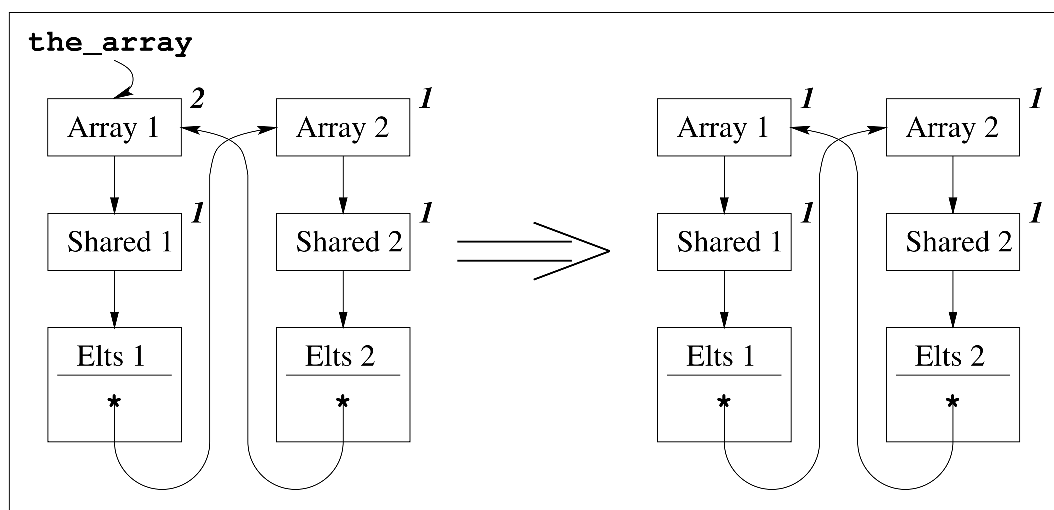
dynamisch geheugen Dynamische geheugen wordt (om het bondig uit te drukken) gebruikt in alle andere gevallen. Dit betekent dat al wat door swapping naar secundair geheugen mag geschreven worden, en al wat slechts en beperkte levensduur heeft, in dynamisch geheugen terecht moet komen.

[5] Dit wordt in engelstalige literatuur een “garbage collector” genoemd.

Tijdens de opstartfase van DGD wordt er met statisch geheugen gewerkt, en daarna zal algemeen dynamisch geheugen gebruikt worden, tenzij expliciet wordt aangegeven dat een allocatie in statisch geheugen moet gebeuren. Deze scheiding van geheugen ontlast de opkuisfunctie omdat het statisch geheugen toch nooit opgekuist mag worden. Het heeft dus geen zin dit telkens weer te controleren. Verder zorgt het er ook voor dat een betere opkuisfunctie geïmplementeerd kan worden. De opkuisfunctie die gebruikt wordt in DGD is niet ideaal, doch voldoet in de praktijk[6].

3.1.2 Opkuisen met verwijzingstellers

DGD implementeerd een opkuisfunctie gebaseerd op verwijzingstellers. Met een gegevenselement wordt een teller geassociëerd, die bijhoudt hoeveel verwijzingen er bestaan naar dat bepaalde element. Indien de teller nul wordt, kan het element opgekuist worden. Meestal wordt het gegevenselement dan op een lijst van vrij te gebruiken elementen gezet waar het gevonden kan worden indien een dergelijk element later weer nodig is.



Figuur 3.1: Problemen met verwijzingstellers

Het is algemeen bekend dat verwijzingstellers niet volstaan indien men cyclische structuren toelaat. Beschouw namelijk figuur 3.1. Hier is duidelijk te zien dat wanneer de laatste verwijzing naar de gegevensstructuur verdwenen is, er toch geen geheugen opgekuist kan worden omdat geen van beide tellers op nul komt. Om dit probleem op te lossen zijn er nog twee andere vormen van opkuisen in DGD aanwezig.

3.1.3 Opkuisen tijdens swapping

Vermits objecten naar secundair geheugen geschreven kunnen worden wanneer ze een tijd niet aktief zijn geweest, kan ook hier een opkuis van geheugen gebeuren. De gegevens-

[6] Informatie betreffende andere opkuistechnieken kan gevonden worden in [Wil92] en [Bjö89].

structuren die door zulk een object worden gebruikt, mogen namelijk vernietigd worden zodra het object op secundair geheugen weggeschreven is. Hier wordt momenteel echter slechts gedeeltelijk gebruik van gemaakt. Het is namelijk zo dat arrays op een speciale manier behandeld worden in DGD. Om het wijzergedrag te kunnen verzekeren tijdens een uitvoeringscyclus (en algemeen ten gevolge van het vroegere gedrag waarbij wijzers naar arrays in andere objecten bleven bestaan totdat swapping optrad), worden arrays in een centrale module van DGD verwerkt. Alle arrays worden uit één globale hoop gehaald, en komen daar ook terug in terecht wanneer alle verwijzingen verdwenen zijn. Dit heeft echter wel tot gevolg dat er arrays zullen bestaan die enkel tijdelijk een nut hebben (bijvoorbeeld tussentijdse waarden). Deze arrays zullen niet met een bepaald object geassocieërd zijn en indien er dus tijdelijke arrays zijn die cyclische verwijzingen bevatten kunnen ze niet vernietigd worden bij swapping.

3.1.4 Opkuisen bij een dump

Gelukkig is er nog een derde fase in de opkuisstrategie van DGD. Na een dump of swapping van *alle* objecten zijn alle overblijvende gegevens in het dynamisch geheugen overbodig. Al wat nog bestaat is "vuilnis" die door de vorige opkuisstrategieën niet kon worden verwijderd. In deze derde fase wordt gewoon al het dynamische geheugen weer vrijgegeven. Dit is toegelaten omdat al de nuttige informatie reeds weggeschreven is naar secundair geheugen.

3.2 Functieoproepen

Zoals reeds vermeld werd op pagina 22 is in de standaard DGD implementatie van LPC een inter–object functieoproep naar een onbestaande of "static" functie geen fout, maar gewoon een oproep die niets doet en 0 als waarde teruggeeft. Er wordt ook aangegeven dat dit gedrag niet gewenst is voor een programmeertaal. Dit kan onder andere leiden tot zeer moeilijk te detecteren fouten in een klasse. Vandaar dat als eerste aanpassing aan DGD de afhandeling van dit soort functieoproepen werd aangepakt. Het bleek ook onmiddellijk de kleinste aanpassing te zijn die voor deze thesis werd uitgevoerd.

Hoezeer dit ook de makkelijkste aanpassing is in DGD, blijkt het ook dadelijk een aanpassing te zijn die het vooropgestelde doel van continue verificatie danig in de war schopt. Aangezien MUDs ontwikkeld zijn met de wetenschap dat een inter–object functieoproep nooit een fout zal geven tenzij de opgeroepen functie een fout bevat, zitten er heel wat functieoproepen in de code naar objecten van klassen die de gewenste functie niet noodzakelijkerwijs bevatten. Dit met als gevolg dat bijna onmiddellijk fouten gedetecteerd worden. Om toch te kunnen testen of de aanpassing juist is, werd een configuratievariabele toegevoegd aan DGD om aan te geven of er strikte controle moet gebeuren op inter–object oproepen, of niet. Met een zeer beperkte verzameling klassen is dan snel te controleren of de fout-detectie juist werkt, en voor het testen van verdere aanpassingen kan dan tijdelijk de strikte controle uitgeschakeld worden.

Een inter–object functieoproep wordt in DGD uitgewerkt als een oproep van een primitieve functie, namelijk de kfun *call_other()*[7]. Dit betekent concreet dat volgende stappen doorlopen worden bij een inter–object oproep (de aanpassingen die voor deze thesis werden uitgevoerd zijn vet gedrukt weergegeven):

1 Roep kfun *call_other(obj, func, args)* op.

1.1 Roep functie *i_call(obj, func, args)* in de vertolker op.

1.1.1 Indien het object nog niet geïnitialiseerd was, doe dit dan nu door de creatie–functie op te roepen.

1.1.2 **Indien *func* leeg is, was dit slechts een oproep om te garanderen dat de creatie–functie werd opgeroepen. Uitvoering van deze routine wordt beëindigd.**

1.1.3 **Indien de functie niet gevonden kan worden in de functietabel van het object *obj*, geef dan een uitvoeringsfout.**

1.1.4 **Indien de functie "static" blijkt te zijn, geef dan een uitvoeringsfout (tenzij de oproep "static" functies toelaat, zoals bijvoorbeeld een *call_other()* naar het oproepende object zelf).**

1.1.5 Roep de functie *func* op in *obj*.

De aanpassing in punt 1(1)2 lijkt misschien ongewoon binnen het kader van de inter–object functieoproepen, maar in feite is het een gevolg van de eigenlijke aanpassing. Bij creatie van een object moet er een creatie–functie uitgevoerd worden. Dit wordt gedaan door een functie zonder naam op te roepen in het object. Bij de standaard DGD implementatie wordt dat gewoon afgewerkt als een oproep van een functie die niet bestaat. Vermits dit echter een fout hoort te geven in deze thesis implementatie, moet er expliciet nagegaan worden of de oproep uitsluitend dient om de creatie–functie op te roepen. Vandaar deze extra aanpassing.

De twee andere aanpassingen (1(1)3 en 1(1)4) implementeren de twee gevallen waarin een uitvoeringsfout moet gegeven worden. In beide gevallen betreft het een simpele test die bij faling de gewenste uitvoeringsfout zal geven.

3.3 Klasse en entiteit concept

Op pagina 14 werd aangegeven dat voor deze thesis een scheiding is ingevoerd in DGD tussen klassen en zogenaamde entiteiten. Een entiteit werd reeds gedefiniëerd als een object waarvan er maar één exemplaar mag bestaan. Het invoeren van deze scheiding leidt uiteraard tot een aanpassing aan DGD. Alvorens deze aanpassing te kunnen bespreken, is enige uitleg nodig betreffende de creatie van objecten.

[7] Inter–object functieoproepen kunnen ook geschreven worden met behulp van de ->operator. Dit zal door de vertaler echter omgezet worden in een *call_other()*, en dus mogen we ons beperken tot die kfun.

Algemeen zijn er slechts twee manieren waarop een object tot stand kan komen. Het zal dadelijk duidelijk worden dat deze methodes zeer specifiek zijn voor een bepaald soort objecten.

1 **Oproep van een functie in klasse of entiteit.**
In dit geval wordt er een poging gedaan de blauwdruk van die bepaalde klasse of entiteit[8] te laden. Indien blijkt dat deze klasse of entiteit niet bestaat, of indien er een fout optreedt tijdens de vertaling, zal er een foutboodschap gegeven worden. Indien de vertaling slaagt, zal de functie opgroepen worden. Deze oproep zal (zoals in het vorige deel werd aangegeven) allereerst een oproep van de creatie–functie tot gevolg hebben, waarna de effectieve oproep zal gebeuren.

   Deze methode is uiteraard enkel bruikbaar voor blauwdrukken.

2 **Maken van een kloon van een klasse.**
Met behulp van de kfun *clone_object()* is het mogelijk een instantiatie te creëren van een klasse. Indien deze klasse nog niet geladen werd, zal ze vertaald worden, waarna er een kloon aangemaakt zal worden (onder voorbehoud van een succesvolle vertaling). In deze kloon zal dan dadelijk de creatie–functie opgeroepen worden.

   Zoals reeds eerder werd vermeld, kan men *clone_object()* niet toepassen op entiteiten. Ook kan men nooit een blauwdruk bekomen met behulp van deze kfun.

Verder kan er tijdens de vertaling van een klasse of entiteit aangegeven worden dat inheritance van een andere klasse moet gebeuren. Dit kan wederom aanleiding geven tot vertaling van die klasse die weer via inheritance van andere klassen kan afhangen. Inheritance van entiteiten is uiteraard verboden.

Concreet werd voor deze thesis dus een syntactische wijziging doorgevoerd aan DGD. Programmacode moet vervat worden in een superstructuur, die ofwel als een klasse wordt gedefiniëerd, ofwel als een entiteit. In de geest van de syntax van C en LPC zelf werd verkozen om de programmacode van een klasse of entiteit te omsluten in accolades, en deze te laten voorafgaan door de vermelding “class” of “entity” gevolgd door een willekeurige string die als naam dient.

Verder werden er aanpassingen uitgevoerd aan de vertaler en aan de implementatie van de kfun *clone_object()* om te verzekeren dat inheritance en klonen van een entiteit niet toegelaten zijn. Tijdens de vertaling is het allereerste element in de beschrijving van de klasse of entiteit juist de specificatie of het om een klasse gaat of een entiteit. Indien het een entiteit betreft (minst voorkomende geval), zal er een vlag gezet worden om aan te geven dat dit een entiteit is. Bij klonen en inheritance wordt deze vlag geraadpleegd.

Het is misschien onlogisch dat het mogelijk is een naam op te geven bij de specificatie van een klasse of entiteit, doch zoals uit het volgende onderdeel zal blijken, is dit ook een aanpassing die werd uitgevoerd. Aangezien beide aanpassingen gelijktijdig ontworpen werden, is hier reeds de voorbereiding ervoor aanwezig.

[8]Hier voeren we de terminologie in dat de blauwdruk van een entiteit, de entiteit zelf is.

3.4 Naamgeving van klassen en entiteiten

Zoals in vorig deel te lezen is (en ook op pagina 14), is er gekozen om klassen en entiteiten een naam te geven. Dit is in ongeveer alle object gerichte talen zo, en er zijn concrete redenen waarom dit een pluspunt is in LPC. Deze verantwoording is zeker nodig omdat de standaard implementatie refereert naar klassen op basis van de bestandsnaam[9]. De aanpassing om arbitraire namen te kunnen gebruiken was erg complex en zonder een goede reden misschien zelfs ongewenst. Zoals dadelijk zal blijken is het voordeel van arbitraire namen zeker voldoende om een uitgebreide aanpassing te verantwoorden.

- Arbitraire namen hebben het voordeel dat ze (meestal) geen informatie bevatten betreffende de plaats waar de programmacode te vinden is. Dit betekent dus dat er een scheiding is tussen de logische en de fysische vorm van een klasse of entiteit. Het grote voordeel hiervan is dat de programmacode van een klasse kan verplaatst worden zonder dat dit een negatieve invloed heeft op andere objecten. Denk namelijk maar aan de gevolgen waar een verplaatsing van een erg cruciale klasse toe kan leiden.
- In dezelfde visie als het voorgaande punt, is het belangrijk in te zien dat DGD werkelijk een globale abstracte wereld implementeerd die erg dynamisch is, voornamelijk ten gevolge van het feit dat de gebruiker de omgeving slechts gebruikt als één van vele gebruikers. De server is ook ontworpen om zonder onderbreking te blijven werken terwijl de gebruikers op willekeurige momenten in de omgeving kunnen werken.
- Het is nooit goed om de gebruiker beperkingen op te leggen tenzij er een goede reden voor bestaat. De verbruiker verplichten een bepaald mechanisme te gebruiken voor de naamgeving van klassen is niet goed. Conceptueel hoort er geen verschil te bestaan tussen bestandsnamen en arbitraire namen, dus zelfs al zou efficiëntie een geldige reden zijn om bestandsnamen te gebruiken, is dit toch nog altijd onvoldoende reden.

Van deze drie argumenten is de belangrijkste wel de scheiding tussen logische en fysische voorstelling van een klasse. Object gerichtheid impliceert een abstractie (zoals vermeld wordt in de definitie op pagina 3). Het gebruik van arbitraire namen versterkt de abstractie door de klassen en entiteiten los te maken van de locatie van hun programmacode. Vermits enkel efficiëtie en complexiteit van de aanpassing als tegenargumenten aan te voeren zijn, is de overschakeling naar arbitraire namen zeker gerechtvaardigd.

Alvorens de effectieve werking van de vertaler te beschrijven, en de aanpassingen nodig om arbitraire namen te gebruiken, is het nodig op te merken dat in het kader van de structuur van DGD deze conceptuele verandering ook intern dient doorgevoerd te worden. Aanpassingen aan een dergelijk programma dienen zoveel mogelijk uitgevoerd te worden volgens de geest van de originele programmeur. Dit mag niet al te slaafs gevolgd worden natuurlijk, maar indien de ideeën achter het originele ontwerp goed zijn, is het best deze te volgen. Hier betekent dit dat de werking van DGD intern sterk overeenkomt met de

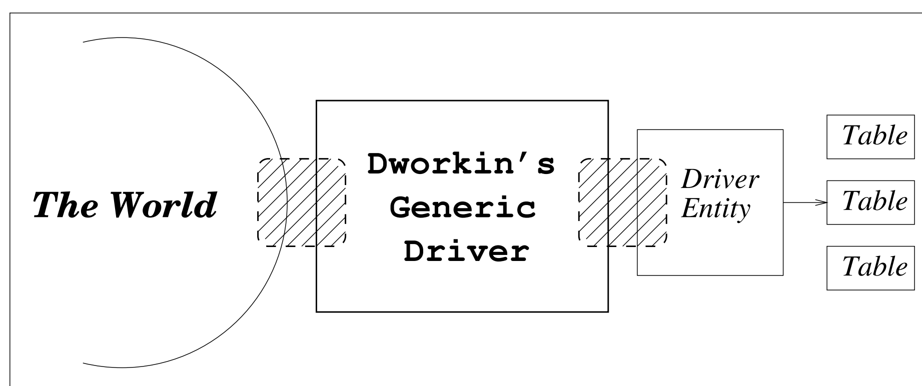
[9]Omdat DGD bijna altijd op Unix–systemen gebruikt wordt, volgen bestandsnamen de syntax die op Unix gebruikelijk is. Bijvoorbeeld: */users/leda/stuffed/bunny.c*

werking van de LPC omgeving. Indien de vertaler een object moet opzoeken in interne structuren worden dezelfde functies gebruikt als welke gebruikt worden indien vanuit LPC een object moet opgezocht worden.

3.4.1 Conversie van arbitraire namen naar bestandsnamen

Om de conversie tussen arbitraire namen en bestandsnamen te kunnen doen, moet er ergens een tabel bijgehouden worden die voor elke arbitraire naam de overeenkomstige bestandsnaam vermeldt. Deze tabel zou in DGD zelf bijgehouden kunnen worden, maar dit gaat in tegen de filosofie dat DGD slechts de basis hoort te zijn van de systeem, en dat enkel het hoogst noodzakelijke aanwezig hoort te zijn in DGD zelf. Daarom is het beter deze informatie in een LPC structuur van objecten bij te houden. Dit is ook heel wat flexibeler.

Omdat DGD ook conversies moet kunnen doen van bestandsnamen, is er een soort interface[10] nodig. De "driver" entiteit vervult deze functie reeds voor andere functies, en dus wordt er voor de naamconversie een functie bijgemaakt[11]. Omdat er een (instelbare) limiet bestaat op de maximale grootte van arrays en mappings, is het niet voldoende één enkele mapping te gebruiken voor de conversietabel. Een meer dynamische structuur is hier aangewezen, en de objecten in die structuur worden dan door de driver entiteit geraadpleegd. Dit laat ook toe om later aanpassingen uit te voeren aan de conversietabel en de algoritmen die gebruikt worden voor raadplegingen.



Figuur 3.2: Naam conversie–mechanisme

Hier zit uiteraard een kip–en–ei probleem verborgen. Vermits de driver entiteit de raadpleging verricht, kan de bestandsnaam van deze entiteit niet op deze manier bekomen worden. Per uitzondering zal deze bestandsnaam dan ook in een bestand met instellingen voor DGD opgenomen worden. In feite bestaat er een gelijkaardig probleem voor de klassen

[10] Hiervoor wordt soms de nederlandstalige term grensvlak gebruikt. De semantische betekenis van de twee termen komt echter niet overeen, en dus lijkt het best de engelstalige term te behouden.

[11] Eigenlijk wordt gewoon de semantiek van één van de functies aangepast. Het blijkt namelijk dat door de naamconversie een functie in de driver entiteit nutteloos wordt.

(en mogelijke generalisaties) die gebruikt worden om de conversietabel voor te stellen. Hiervoor moet er een kleine, vaste conversietabel opgenomen worden in de driver entiteit. Figuur 3.2 geeft een grafische voorstelling van het conversie–mechanisme.

3.4.2 Het vertalingsproces

Wat ook de reden voor vertaling mag zijn (oproep naar een ongeladen blauwdruk, oproep van *clone_object()* voor een ongeladen klasse, ...), steeds zal een centrale functie in de de vertaler opgeroepen worden, namelijk *c_compile(naam, bestand)*. Bij vertaling van een klasse die voor inheritance nodig is, wordt deze functie *niet* opgeroepen. In dat geval begint de vertaling bij punt 2 in het vertalingsalgoritme dat verder gegeven zal worden. Dergelijke vertalingen volgen uiteindelijke hetzelfde algoritme, maar beginnen één stap verder zoals later zal blijken. De centrale vertalingsfunctie krijgt volgende gegevens:

naam Dit is de arbitraire naam van de klasse (of entiteit) zoals deze werd opgegeven bij de *clone_object()* invocatie, of bij de functieoproep naar de blauwdruk.

bestand Dit is de naam van het bestand waarin de vertaler de programmacode voor de klasse of entiteit kan vinden.

Er kan al dadelijk discussie ontstaan betreffende wanneer de bestandsnaam opgevraagd moet worden. Dit kan in de oproepende LPC code gebeuren, of bij de oproeper van de vertaler, of in de vertaler zelf. Het is duidelijk dat het niet gewenst is dat de bestandsnaam reeds op LPC niveau bekomen dient te worden. Dat zou namelijk de doelstelling van deze aanpassing teniet doen. De vertaler zelf lijkt onmiddellijk de beste plaats te zijn, omdat die centraal is voor alle oproepers. Het is echter zo dat de kfun *call_other()* normaal een fout geeft indien het eerste argument (het doelobject) niet bestaat. De standaard implementatie van DGD gaf dezelfde fout ook indien er een string werd opgegeven die onmogelijk een bestandsnaam kon zijn[12]. Om dit gedrag te volgen, moet de conversie van arbitraire naam naar bestandsnaam reeds bij de oproeper van de vertaler gebeuren, zodat indien de conversie niet succesvol is, een fout kan gegeven worden.

De vertaling van een klasse of entiteit verloopt volgens een iteratief algoritme omdat de vertaling via inheritance van andere onvertaalde klassen kan afhankelijk zijn. het algoritme is als volgt (het wordt gegeven voor klassen, maar uiteraard is het ook geldig voor entiteiten):

1 Roep functie *compile(naam, bestand, 0)* op.

2 Voer volgende semi–oneindige lus uit:

[12]De kfun *call_other()* kan als eerste argument zowel een string als een object aanvaarden. Indien het een string is, dan wordt er een object dat dezelfde stringrepresentatie heeft als de gegeven string. Wordt er zo geen object gevonden dan wordt er geprobeerd de string te interpreteren als een bestandsnaam. Dit klinkt onlogisch, maar het formaat van de representatie verklaart dit: “bestandsnaam#nummer”

2.1 Indien het bestand niet de driver entiteit is, en het ook niet de vertaling betreft van de object klasse:

2.1.1 Indien de driver entiteit niet geladen is, ga dan naar punt 2 met naam ≡ lege string, en bestand ≡ bestand met de driver entiteit.

2.1.2 Indien de object klasse niet beschikbaar is, ga dan naar punt 2 met naam ≡ naam van de object klasse, en bestand ≡ bestand dat hierbij hoort.

2.2 Indien het bestand niet gevonden kan worden:

2.2.1 Indien het niet de driver entiteit betreft, en ook niet de object klasse:

2.2.1.1 Doe oproep naar *driver->compile_object(naam).*

2.2.1.2 Indien de oproep succesvol is, geeft aan het bekomen object de gegeven naam.
Verlaat de lus.

2.2.2 Geef een vertalingsfout.

2.3 Indien het standaard definitie bestand[13] niet kan gelezen worden, geef dan een vertalingsfout.

2.4 Begin de vertaling:

2.4.1 Lees invoer uit het bestand totdat het einde bereikt is, en doe ondertussen:

2.4.1.1 Haal een invoerblokje door de lexische analyse.

2.4.1.2 Haal het resultaat van de lexische analyse door de syntactische analyse.

2.4.1.3 Verwerk het resultaat van de syntactische analyse (dit is meestal generatie van uitvoerbare code).

2.5 Indien de vertaling successvol is, creëer dan een nieuw object.
Verlaat de lus.

2.6 Indien de vertaling werd afgebroken wegens nood aan een klasse die wegens inheritance nodig is, ga dan naar punt 2 met naam ≡ naam zoals die bij de inherit instructie werd gegeven, en bestand ≡ de bestandsnaam die erbij hoort.

2.7 De vertaling geeft een fout indien dit punt bereikt wordt.

In punt 2(1)2 moet er een conversie gebeuren van de naam van de object klasse naar de bestandsnaam die erbij hoort. Ook in punt 2(4)13 kan dit nodig zijn, namelijk wanneer een inherit instructie moet verwerkt worden. Vermits de object klasse ook via inheritance in de te vertalen klasse moet opgenomen worden, is het duidelijk dat de functies die inheritance implementeren de beste plaats zijn om de conversie uit te voeren. Dit komt neer op een redelijk eenvoudige aanpassing. Bij inheritance moet nagegaan worden of de klasse reeds vertaald is geworden, en indien niet moet deze vertaald worden alvorens de vertaling van de huidige klasse voort te kunnen zetten. Als de klasse dus niet gevonden kan worden in de lijst van aktieve klassen (dit zoeken gebeurt op basis van de klasse naam), wordt de naam geconverteerd naar de bijhorende bestandsnaam.

[13] Dit is een vertaling uit het engels. Eigenlijk is “include file” duidelijker, maar vermits overal “bestand” wordt gebruikt in plaats van “file”, lijkt het geen goed idee de engelse term hier te gebruiken.

3.4.3 Nevereffecten

Vermits DGD zelf heel wat objectmanipulaties moet doen (voornamelijk ten gevolge van LPC instructies die in de vertolker worden uitgevoerd), moet ook een keuze gemaakt worden betreffende de basisnaam voor een object. In de standaard implementatie is dit de bestandsnaam en in deze thesis is dit de klasse naam. Het is duidelijk dat deze verandering een enorme invloed moet hebben op het gehele DGD programma, aangezien bijna overal met objecten gewerkt wordt. Dit volgt uiteraard uit het feit dat de voornaamste taak van DGD bestaat uit het vertolken van LPC code. De omvang van deze aanpassing leidt dan ook tot vragen betreffende het nut van een globale aanpassing. Was het niet beter geweest enkel in LPC met arbitraire namen te werken, en bij de overgang naar uitvoering in DGD (dus in de vertolker) waar nodig gewoon de conversie uit te voeren?

Het antwoord op deze vraag dient kort en bondig gegeven te worden als: "Neen!" Reden hiervoor is vrij triviaal. De conversie van namen gebeurt door middel van LPC objecten. Dit betekent dat zolang deze objecten zelf niet geladen zijn, er geen conversie kan plaatsvinden. In een vorige deel werd reeds aangegeven dat dit deels kan opgelost worden met een statische conversietabel in de driver entiteit. Dit volstaat echter niet indien DGD zelf nog met bestandsnamen werkt, omdat niet enkel de conversie aangetast wordt, maar ook de uitvoering van heel wat primitieve functies in LPC. Alle functies die een klasse naam als argument aanvaarden kunnen niet gebruikt worden totdat de conversieobjecten in het systeem aanwezig zijn. Dit is een beperking die zeer zwaar is, en die zelfs fataal kan blijken. Ook legt het een beperking op aan het gebruik van functies die naderhand blijkt te verdwijnen. Een dergelijke anomalie is zeer zeker niet positief, en het alternatief (een grotere aanpassing aan DGD) is zeker de moeite waard, want het geeft:

- Een proper concept voor de werking van DGD.
- Een gelijkaardige werking voor zowel LPC als DGD zelf.
- Grotere flexibiliteit voor de gebruikers.
- Eén enkel concept voor de naamgeving van klassen en entiteiten.

3.4.4 Implementatiedetails

De uiteindelijke implementatie is geïntegreerd in de bestaande code van de vertaler in DGD. In verschillende modules waren aanpassingen nodig omdat er zo algemeen met objecten gewerkt wordt. Allereerst moest natuurlijk de module die objecten beheert aangepast worden. Voor de ondersteuning van arbitraire namen is het namelijk niet voldoende om in plaats van de bestandsnaam de klassenaam op te nemen in interne tabellen, want de bestandsnaam hoort ook bijgehouden te worden, althans voor testen. Verder is er in de object module en in de implementatie van primitieve functies een impliciete veronderstelling dat de naam van een object de bestandsnaam is. Er wordt namelijk een "/" voor de naam geplaatst alvorens deze door te geven aan de LPC omgeving. Dit is natuurlijk niet gewenst wanneer arbitraire namen gebruikt worden.

De grootste aanpassing voor het invoeren van arbitraire namen bleek de vertaler te zijn. De complexe aard ervan was niet geheel een gevolg van de complexiteit van de vertaler, maar ook een gevolg van de stricte eisen die nageleefd moeten worden bij een dergelijke aanpassing. Het eerder vermelde kip–en–ei probleem mag bijvoorbeeld geen aanleiding geven tot een specialisatie van code om met de bestandsnaam van de driver entiteit te kunnen werken waar voor andere objecten de klassenaam gebruikt wordt. Dit betekent dat, hoewel de bestandsnaam nodig is om het systeem te kunnen starten, zodra de driver entiteit geladen is, er naartoe verwezen moet worden op basis van de entiteitsnaam.

Een ander probleem is dat de object klasse geladen moet zijn alvorens een klasse vertaald kan worden[14]. De vertaling van de object klasse is echter speciaal in de zin dat (zoals in het vertalingsalgoritme op pagina 44 te zien is) allerlei testen niet mogen gebeuren. Deze testen waren normaal makkelijk te doen op basis van de bestandsnaam, doch dit is niet meer mogelijk bij de aangepaste versie. Uit oogpunt van schoonheid en consistentie werd gekozen enkel de klassenaam van de object klasse op te nemen in het configuratie bestand. Dit betekent dat de naam geconverteerd moet worden naar een bestandsnaam met behulp van een functie in de driver entiteit. Deze conversie is niet tijdig genoeg te doen zonder het vertalingsalgoritme danig te herwerken, en na enige pogingen om dit toch te doen, werd besloten de meest triviale oplossing te kiezen door expliciet aan te geven aan de vertaler dat de object klasse op het punt staat vertaald te worden (door middel van een functie argument). Heel wat tijd werd besteed aan het zoeken van meer impliciete methodes om dit probleem op te lossen doch er werd geen enkele oplossing gevonden die voldoet. Hieruit blijkt maar weer dat een mooie oplossing niet altijd mogelijk is, of zelfs wenselijk is.

Doordat in de aangepaste syntax van LPC de specificatie betreffende klasse of entiteit helemaal in het begin van de broncode staat, kan zodra deze allereerste informatie verwerkt is door de syntactische analyse, de arbitraire naam aan de vertaler gemeld worden. Vanaf dat moment kunnen alle testen uitgevoerd worden op basis van die naam, en dient de bestandsnaam enkel nog om te vermelden bij vertalingsfouten, en om op te nemen in de interne tabellen. Dit laatste is in feite niet strict noodzakelijk, maar het hielp heel veel bij het testen van de aanpassingen en vermits het eigenlijk geen moeite kost deze informatie te bewaren (enkel een beetje meer geheugenverbruik) werd verkozen dit extraatje niet te verwijderen.

3.5 De wijzer–type anomalie

In het vorige hoofdstuk werd reeds een hele discussie gewijd aan het probleem van de wijzer–type anomalie. Op pagina 27 is de bepreking te vinden van dit probleem. Zoals reeds aangegeven werd is er enkel een probleem ten gevolge van de swapping die in DGD gebeurt. Het lijkt dan ook erg interessant na te gaan of misschien de swapping kan vermeden worden. Dat zou het probleem al onmiddellijk oplossen.

Het blijkt echter dat swapping de basis is waarop de continuïteit van DGD steunt. Enkel door de swapping is het mogelijk het geheugengebruik beperkt te houden door secundair

[14] Het enige object dat vertaald kan worden zonder de object klasse is de driver entiteit.

geheugen te gebruiken om het primaire geheugen te ontlasten. Ook blijkt dat het swapping mechanisme belangrijk is voor het opkuisen van geheugen. Een globale swapping van de LPC omgeving laat toe om achtergebleven vuilnis op te halen, zodat alle geheugen terug beschikbaar komt. Dit toont aan dat swapping eignelijk niet uit DGD genomen mag worden, indien men het concept van de continuïteit belangrijk vindt. Op pagina 34 werd aangegeven dat dit concept net één van de belangrijkste redenen is om LPC en DGD te gebruiken, en zodoende kan de verwijdering van swapping niet aanvaard worden.

Er is dus een andere oplossing noodzakelijk. Op pagina 28 werden een aantal mogelijke oplossingen beschreven, en werd reeds aangegeven dat slechts één van deze oplossingen aanvaardbaar is. Op zich is de gegeven oplossing, namelijk het forceren van het maken van copieën van geïmporteerde[15] arrays en mappings, redelijk triviaal. Het klinkt als een logische oplossing en lijkt zelfs helemaal niet complex. In praktijk blijkt het probleem echter groter te zijn dan men zou vermoeden. Arrays vormen een erg complex deel van DGD, ten gevolge van hun centraal beheer en het feit dat deze gegevensstructuren tot de wijzer–types behoren.

3.5.1 Wijzer–types intern in DGD

Arrays en mappings (de enige wijzer–types in LPC[16]) worden intern uitgewerkt met een indirecte verwijzing. Op pagina 24 werd dit reeds vermeld in functie van het mechanisme voor parameterbinding. Er zijn echter diepere redenen voor deze indirectie. Beschouw namelijk figuur 3.3, waarin een fictieve situatie wordt geschetst met directe referenties voor mappings[17]. Zoals duidelijk te zien is, veroorzaakt dit problemen indien een mapping waarnaar twee verwijzingen bestaan, veranderd wordt.

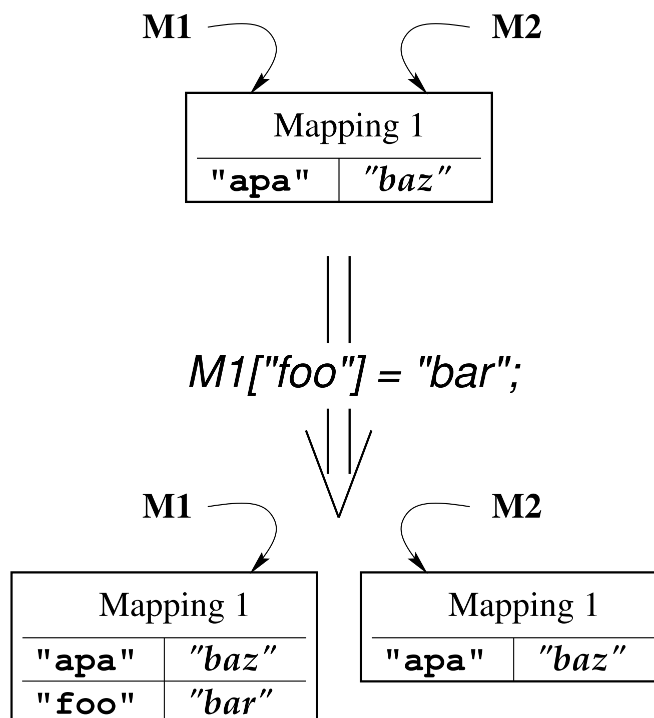
Het is namelijk zo dat voor de toekenning van "bar" aan het element "foo" in M1, de twee mapping variabelen M1 en M2 gelijk zijn. Na de toekenning is dit niet meer het geval! De toekenning voegt namelijk een element toe aan de mapping, en vermits slechts geheugen voor één element gealloceerd werd, moet er een nieuw geheugengebied aangemaakt worden dat twee elementen kan bevatten. M2 blijft echter naar het oude gebied wijzen. Dit is een gedrag dat niet overeenkomt met de definitie van mappings in LPC, en dus moet er een extra niveau van wijzers ingevoerd worden om te zorgen dat verwijzingen naar een mapping steeds gelijk blijven. Het enige dat zal veranderen is het geheugengebied waarin de elementen opgeslagen worden, en daar bestaat maar één verwijzing naar.

Het doel is om op het einde van een uitvoeringscyclus te verzekeren dat alle arrays en mappings waarnaar verwezen wordt vanuit een object ook tot dat object zelf behoren en niet tot een ander object. Anderzijds is het de bedoeling dat verwijzingen naar arrays

[15]We beschouwen een array geïmporteerd indien het toegekend is geworden aan een globale variabele in het doelobject. Merk op dat zulk een toekenning ook onrechtstreeks kan gebeuren. Een toekenning aan een element in een array in het doelobject is ook een import.

[16]In feite zijn strings ook wijzer–types voor wat DGD betreft. Het gemeenschappelijke aspect is dat het gegevenstypes zijn met een dynamische grootte. Vermits strings echter niet gedeeld worden tussen objecten, beschouwen we het niet als een wijzer–type.

[17]Intern worden mappings geïmplementeerd op basis van arrays.



Figuur 3.3: Illustratie van directe verwijzingen naar mappings

en mappings in andere objecten wel mogelijk zijn binnen éénzelfde uitvoeringscyclus. Dit betekent dat er een manier moet zijn om na te gaan of een array tot een bepaald object behoort. Objecten worden in het geheugen voorgesteld met een "control block"[18] en een "dataspace"[19]. Hieruit kan afgeleid worden dat het volstaat een array te associëren met een dataspace in plaats van met een object (dit lijkt misschien onnodig, maar intern in DGD is dit de meer logische keuze).

In de standaard implementatie van DGD bestaan er 2 soorten arrays:

1 **Vlottende arrays**
Dit zijn arrays die (nog) niet geassocieerd zijn met een bepaalde dataspace. Tijdelijke waarden, of arrays die enkel aan lokale variabelen toegekend zijn geworden behoren toe aan deze soort. Arrays die toegekend zijn aan globale variabelen in een array zijn vlottende arrays totdat swapping van de dataspace gebeurt.

2 **Arrays in eigendom**
Zodra een array toegekend wordt aan een dataspace, en deze dataspace door swapping naar secundair geheugen wordt geschreven, wordt dit array een array in eigendom. Bij het herinladen van de dataspace van het object zullen deze arrays namelijk in een

[18]Dit is de term die in DGD gebruikt wordt om het geheugengebied aan te geven dat informatie bevat over het programma dat de klasse beschrijft waarvan het object een instantiatie is.

[19]Deze term wordt gebruikt om het geheugengebied te beschrijven dat alle lokale gegevens bevat van een object. Voornamelijk de globale variabelen in het object, en nog wat bijhorende informatie.

apart geheugenblok opgeslagen worden. Het beschrijvingsblok van een array bevat echter geen informatie betreffende de eigenaar ervan.

3.5.2 Eerste mogelijke implementatie

Het onderscheid tussen de twee soorten arrays is theoretisch gezien niet nodig om het vereiste copiëermechanisme te implementeren. Het is echter nodig om in dat geval alle variabelen in de dataspaces af te gaan om te controleren of het arrays[20] zijn, en dan de hele grafe af te gaan die met dat array geassociëerd is. Met een markeringsalgoritme kan dan verzekerd worden dat elk array slechts één maal verwerkt wordt. Dit algoritme is reeds in DGD aanwezig om bij swapping te vermijden dat een array meerdere malen weggeschreven wordt.

Er zijn een aantal problemen met deze aanpak. Allereerst moet de hele dataspace van alle aanwezige objecten afgegaan worden (sinds de laatste swapping van de dataspace kunnen er arrays bijgekomen zijn in een object indien er variabelen van het mixed type bestaan, en die bevinden zich niet in de lijst van arrays in eigendom). Dit is een heleboel werk dat eigenlijk onnodig is. Efficiëntie is zelden een goede reden om een oplossing af te wijzen, maar een aanpassing van deze aard is belangrijk, en moet zeer dikwijls uitgevoerd worden. Een meer efficiënte aanpak is daarom zeker een pluspunt. Vandaar dat de triviale oplossing hier niet aanvaardbaar is.

Merk op dat arrays waarnaar slechts één verwijzing bestaat niet gecopiëerd hoeven te worden. Dergelijke arrays kunnen slechts tot één enkele dataspace behoren (die waar de ene verwijzing in te vinden is).

3.5.3 Tweede mogelijke implementatie

Als oplossing voor het onnodige werk dat het eerste implementatievoorstel als onaanvaardbaar bestempelt kan een expliciete lijst bijgehouden worden van alle arrays die geïmporteerd worden in dataspaces. Enkel deze arrays moeten behandeld worden (en natuurlijk ook de grafe van verwijzingen naar andere arrays die er als elementen in kunnen voorkomen). Dit stelt wel een probleem: “hoe kan bepaald worden dat een array geïmporteerd wordt?” Dit impliceert drie gevallen:

1 Een array is vlottend.

2 Een array behoort tot een andere dataspace.

3 Een array werd reeds eerder geïmporteerd (maar wel binnen dezelfde uitvoeringscyclus omdat anders het array reeds tot de dataspace behoort). Een array hoort enkel bij de allereerste import in de lijst van geïmporteerde arrays opgenomen te worden.

[20]In dit hoofdstuk zullen enkel arrays verder besproken worden. Vermits mappings op basis van arrays geïmplementeerd zijn, moet enkel met het globale concept “array” rekening gehouden worden.

Een array is vlottend indien het nog niet werd toegekend aan een globale variabele in een dataspace. Dit zou gecombineerd kunnen worden met het aspect van eigendom van een array, namelijk door een vlottend array eigenaar NULL[21] te geven. Echter... enerzijds is het niet goed te verantwoorden dat een array *geen* eigenaar zou hebben. Er is namelijk altijd een bepaald object verantwoordelijk voor de creatie van een array. Anderzijds is het niet goed twee concepten met elkaar te verweven. Wat dus nodig is, is een soort vlag om aan te geven dat een array vlottend is, en een verwijzing naar de dataspace die eigenaar is van het array.

Het laatste geval is moeilijker. Het is namelijk zo dat voor elk array moet bijgehouden worden in welke dataspaces het geïmporteerd. Vermits een array geen kennis heeft betreffende mogelijke andere arrays die elementen hebben die wijzen naar het array, moet het copiëermechanisme beginnen bij een bepaald array, de grafe aflopen die erbij hoort, en dan een volgend array nemen dat nog niet verwerkt werd. Probleem hierbij is voornamelijk het beheer van verwijzingstellers. Er moet bijgehouden worden hoeveel verwijzingen er zijn vanuit een bepaalde dataspace[22]. Indien deze teller namelijk 0 wordt, dient de dataspace verwijderd te worden uit de importlijst. Het beschrijvingsblokje van het array moet natuurlijk het totaal van de tellers in de importlijst als verwijzingsteller bijhouden.

Het is duidelijk dat deze oplossing redelijk complex is, en dat het beheer van de verwijzingstellers ingewikkeld wordt. Uit literatuur betreffende algoritmes voor het opkuisen van geheugen (bijvoorbeeld [Bjö89]) blijkt dat één van de grote nadelen van dergelijk algoritmes net ligt in het feit dat de vertraging die volgt uit het ophogen en verminderen van de verwijzingstellers merkbaar kan worden. Het is dus niet gewenst om een dergelijk belangrijk aspect nog storender te maken. Vandaar dat ook deze oplossing niet werkbaar is.

3.5.4 Derde mogelijke implementatie

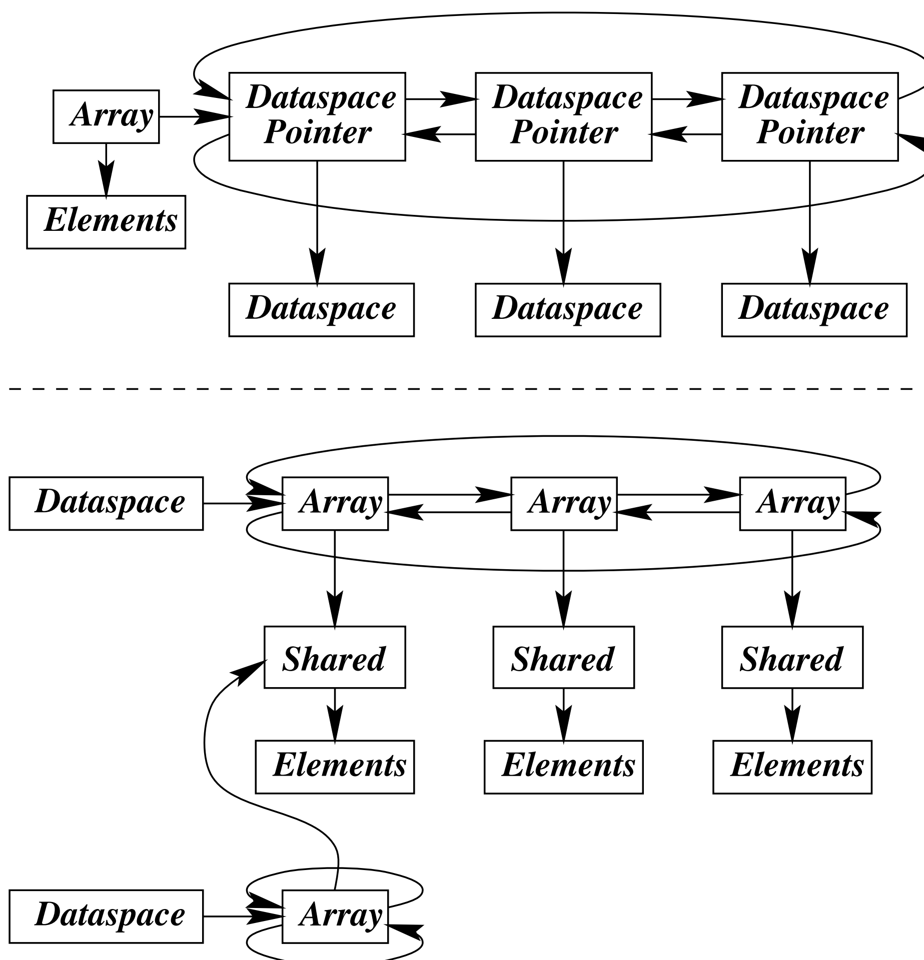
Er is dus nog een betere methode nodig om tot een goede oplossing van het wijzer–type probleem te komen. In plaats van bij een array de dataspaces op te sommen waarin het geïmporteerd is, kunnen de geïmporteerde arrays opgesomd worden bij de dataspaces. Dit betekent dat in plaats van een lijst van verwijzingen naar dataspaces bij een array, een lijst van verwijzingen naar arrays bij een dataspace zal toegevoegd worden. In figuur 3.4 wordt dit verschil grafisch voorgesteld. Ook wordt er reeds aangegeven dat er een terminologieverandering is doorgevoerd. Vermits arrays geïmporteerd worden, en er een deel van het array gedeeld wordt met mogelijke andere verwijzingen (enerzijds wegens import in andere dataspaces, en anderzijds de verwijzing vanuit de eigenaar), wordt dat deel een *gedeeld array*[23] genoemd.

Er zijn voornamelijk twee redenen voor het invoeren van de gedeelde arrays. Allereerst

[21]NULL is in C de standaardwaarde om aan te geven dat een wijzer geen waarde heeft. Dit wordt meestal gedaan om het mogelijk te maken hierop te testen.

[22]Arrays die de dataspace als eigenaar hebben, en die als element een wijzer bevatten naar het beschouwde array tellen ook in de verwijzingsteller voor die dataspace.

[23]In de DGD broncode en in grafische voorstellingen worden deze gedeelde arrays "shared" genoemd.



Figuur 3.4: Tweede (bovenaan) vs derde (onderaan) implementatie

worden arrays bij toekenning aan een globale variabele in een vreemd object (dus het array wordt geïmporteerd) eigendom van die dataspace. Dit zou doen vermoeden dat een array het eigendom kan zijn van meerdere dataspaces. Dit moet echter genuanceerd worden. Het geheel van een gedeeld array samen met de elementen van het array is altijd gelijk voor alle dataspaces die een verwijzing naar het array hebben, en is nooit eigendom van een dataspace. Het beschrijvingsblok van het array heeft echter wel een eigenaar, en na toekenning van een array aan een variabele in een dataspace hoort die dataspace de eigenaar te zijn van dat array. Anderzijds is het zo dat er bij het copiëermechanisme toch een beschrijvingsblok moet zijn in elke dataspace die het array importeerd (en ook in de originele eigenaar). Vermits zulk een exemplaar per dataspace reeds nuttig bevonden is, kan dit enkel een voordeel zijn.

3.5.5 Verdere uitwerking van de implementatie

Met de toevoeging van gedeelde arrays, en de lijst van geïmporteerde arrays per dataspace is er een voldoende basis om het copiëermechanisme verder uit te werken tot een effectieve implementatie. Het is duidelijk dat het een verspilling van tijd zou zijn om alle dataspaces af te gaan op zoek naar geïmporteerde arrays. Vandaar dat dergelijke dataspaces in een dubbel gelinkte circulaire lijst opgenomen zullen worden. Elke dataspace hoort maar één keer opgenomen te worden per uitvoeringscyclus, en indien de arrays die werden geïmporteerd allemaal hun verwijzing vanuit de dataspace mochten verwijderd zien, moet de dataspace uiteraard uit de circulaire lijst verwijderd worden. Een andere reden om verwijderd te worden uit de lijst is natuurlijk een mogelijke vernietiging van het bijhorende object.

In detail zijn volgende manieren mogelijk om een array te importeren:

1 Toekenning van een array uit een andere dataspace aan een globale variabele in de dataspace van het doelobject.

2 Toekenning van een array uit een andere dataspace aan een element in een array dat behoort tot de dataspace van het doelobject.

Een toekenning van een vlottend array aan een globale variabele of aan een element van een array in de dataspace van het doelobject wordt *niet* als import geïnterpreteerd. Als eigenaar wordt gewoon de dataspace van het doelobject genomen, en het array is vanaf dat moment niet meer vlottend.

Na de uitvoering van LPC code in een bepaalde uitvoeringscyclus wordt een functie opgeroepen in de dataspace module om alle dataspaces die geïmporteerde arrays hebben door het copiëermechanisme te halen. Dit mechanisme bevat een aantal algoritmes die eerst kort besproken zullen worden om nadien met een voorbeeld geïllustreerd te worden.

Algoritme Import

Dit algoritme wordt opgeroepen voor alle dataspaces die geïmporteerde arrays hebben.

1 Plaats alle arrays die vast eigendom[24] zijn van de dataspace in de unificatietabel om duplicaten te vermijden.

2 Voor alle arrays die in de import–lijst staan:

  2.1 Indien het gedeeld array van dit array nog niet verwerkt werd, roep dan algoritme Import–Array op voor het gegeven array en de gegeven dataspace.

Algoritme Import–Array

Hier begint het recursief mechanisme dat een copie moet maken van een array–grafe. Het wordt opgeroepen met het array dat het startpunt vormt voor de grafe, en met de dataspace waartoe de copie zal behoren.

1 Indien de verwijzingsteller van het gedeeld array dat bij het gegeven array hoort een waarde groter dan 1 heeft:

  1.1 Creëer een nieuw gedeeld array met een elementenverzameling die even groot is als het gegeven array.

  1.2 Roep algoritme Copy–Values op met als argumenten de elementen van het gegeven array, de elementenverzameling van het nieuw gedeeld array, de grootte van de verzamelingen, en de gegeven dataspace.

  1.3 Verwijder de verwijzing van het gegeven array naar het oud gedeeld array.

  1.4 Verwijs in het gegeven array naar het nieuw gedeeld array.

  1.5 Geef als resultaat het array, en beëindig dit algoritme.

2 Indien de gegeven dataspace niet de eigenaar is van het geven array:

  2.1 Creëer een nieuw array, met een nieuw gedeeld array en een elementenverzameling die even groot is als het gegeven array.

  2.2 Verklaar de gegeven dataspace eigenaar van het nieuw array.

  2.3 Roep algoritme Copy–Values op met als argumenten de elementen van het gegeven array, de elementenverzameling van het nieuw array, de grootte van de array, en de gegeven dataspace.

  2.4 Geef als resultaat het nieuw array, en beëindig dit algoritme.

3 Roep algoritme Flush–Values op met als argumenten de elementen van het array, de grootte van het array, en de gegeven dataspace.

4 Geef als resultaat het gegeven array.

[24]Dit betekent dat deze arrays vroeger reeds gecopiëerd werden.

Algoritme Import–Array(bis)

Algoritme Import–Array blijkt soms tot problemen te leiden, namelijk indien er voor een bepaalde geïmporteerde array–grafe geen verwijzing meer bestaat vanuit de dataspace waartoe het array waaraan de grafe hangt behoort. Dit is een gevolg van het feit dat indien er slechts één verwijzing blijkt te bestaan naar een gedeeld array waarnaar het geïmporteerd array verwijst, er geen copie wordt gemaakt. Dit kan leiden tot arrays waarnaar geen verwijzingen meer bestaan, die toch niet vrijgegeven worden.

De oplossing is redelijk eenvoudig. Het juiste gedrag is dat zelfs indien de verwijzing van het geïmporteerd array de laatste is, er een copie *moet* gemaakt worden. Ook bleek dat mappings problemen kunnen veroorzaken ten gevolge van hun interne representatie. Algoritme Import–Array wordt dan ook aangepast:

1 Indien het een mapping betreft, zorgt dan dat deze in stricte array voorstelling staat[25].

2 Indien de gegeven dataspace niet de eigenaar is van het gegeven array:

  2.1 Creëer een nieuw array, met een nieuw gedeeld array en een elementenverzameling die even groot is als die van het gegeven array.

  2.2 Geef aan dat de elementen in de gegeven dataspace te vinden zijn.

  2.3 Verklaar de gegeven dataspace eigenaar van het nieuw array.

  2.4 Registreer een verwijzing naar dit nieuw array.

  2.5 Roep algoritme Copy–Values op met als argumenten de elementen van het gegeven array, de elementenverzameling van het nieuw array, de grootte van de array, en de gegeven dataspace.

  2.6 Geef als resultaat het nieuw array, en beëindig dit algoritme.

3 Creëer een nieuw gedeeld array met een elementenverzameling die even groot is als die van het gegeven array.

4 Roep algoritme Copy–Values op met als argumenten de elementen van het array, de nieuwe elementenverzameling, de grootte van het array, en de gegeven dataspace.

5 Geef aan dat de elementen in de gegeven dataspace te vinden zijn.

6 Verwijder de verwijzing naar het oud gedeeld array.

7 Verwijs in het gegeven array naar het nieuw gedeeld array.

8 Geef als resultaat het array.

[25]Mappings worden intern voorgesteld met arrays, en indien nodig met een hashtabel voor nieuw toegevoegde elementen. Deze stap voegt deze twee vormen samen tot een enkel array.

Algoritme Copy–Values

In dit algoritme wordt de ene elementenverzameling gecopiëerd naar de andere. Ondertussen worden array (of mapping) elementen aangepast. het algoritme is als volgt:

- Voor alle elementen in de elementenverzameling (dit aantal wordt als parameter aan dit algoritme doorgegeven):

  1 Indien het element een array of een mapping is:

  1.1 Indien het gedeeld array van dit array nog niet verwerkt werd:

  1.1.1 Roep het algoritme Import–Array op voor het gegeven element en de gegeven dataspace, en voeg het resultaat toe aan de unificatietabel.

  1.2 Anders:

  1.2.1 Unificeer het array en voeg een verwijzing toe naar het resultaat van de unificatie.

  1.3 Ga verder met het volgende element.

  2 Indien het element een string is, voeg dan een verwijzing toe aan de string, en ga verder met het volgende element.

  3 Elementen met een ander type worden gewoon gecopiëerd.

Informatie betreffende de recursie

Met behulp van bovenstaande algoritmes wordt de hele grafe doorlopen die aan een gegeven array hangt. Tijdens deze doortocht zal voor elk array dat zich in de grafe bevindt een keuze gemaakt worden:

- Voor arrays die reeds in de unificatietabel staan, wordt de verwijzing aangepast zodat ze wijst naar het array dat in de tabel staat (dit om te verzorgen dat arrays die naar een zelfde gedeeld array wijzen met elkaar geünificeerd worden).
- Arrays die tot een andere dataspace behoren worden gecopiëerd.
- Alle andere arrays zijn eigendom van de dataspace, en hun gedeeld array zal gecopiëerd worden naar de dataspace.

3.5.6 Voorbeeld

Om het copiëermechanisme te verduidelijken is het best een voorbeeld te nemen dat alle bovenstaande gevallen nodig heeft. Het werd uitgewerkt in een versie van LPmud die werd aangepast om op de DGD implementatie voor deze thesis te werken. Rechtstreeks geïmporteerde arrays komen in het voorbeeld niet voor om enerzijds de uitgebreidheid te beperken, en anderzijds omdat deze gevallen triviaal zijn.

Er zijn drie entiteiten in het voorbeeld:

```
entity "Village church" {
    mixed the_array;

    create() {
        the_array = ({ ({}), ({ 0, 1, 2, 3, 4 }), ({}) });
    }

    export_array() {
        return the_array;
    }
}
```

Figuur 3.5: Broncode van Village church

```
entity "Village green" {
    mixed the_array;

    create() {
        the_array = ({ 'a', 'b', 'c' });
    }

    export_array() {
        return the_array;
    }
}
```

Figuur 3.6: Broncode van Village green

```
entity "Village track" {
    doit() {
        mixed a1, a2, ar;

L1:     a1 = "village church"->export_array();
L2:     a2 = "village green"->export_array();
L3:     ar = ({ 3, 2, 1 });
L4:     a2[0] = a1;

L5:     a1[0] = ar;
L6:     a1[1] = a2;
L7:     a1[2] = ar;
    }
}
```

Figuur 3.7: Broncode van Village track

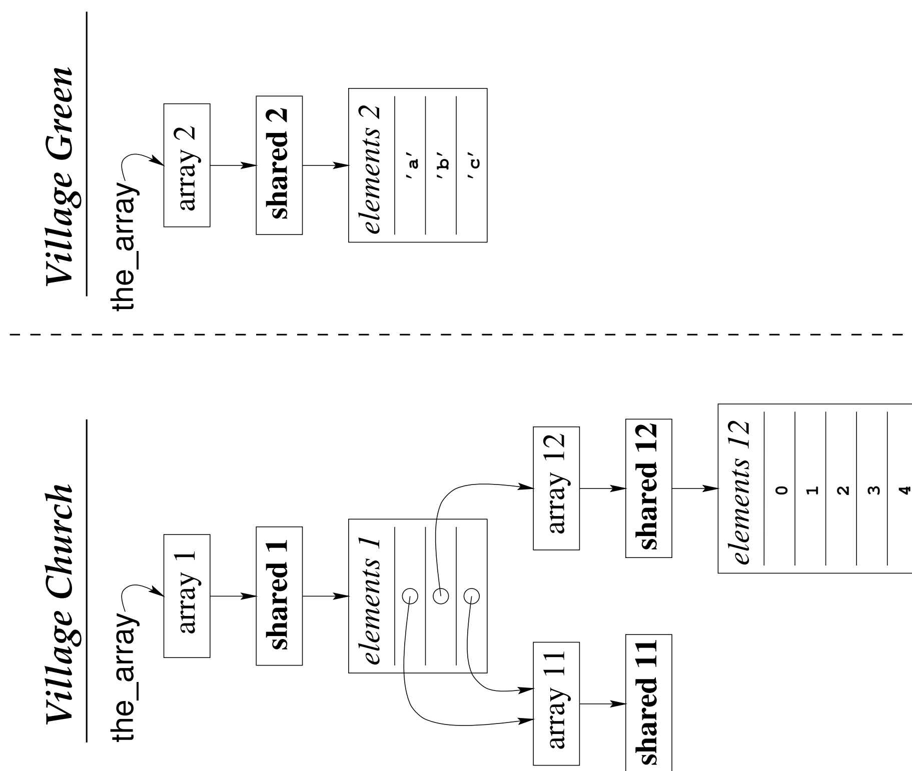
1 **Village church**
Deze entiteit bevat een globaal array ({ ({}), ({ 0, 1, 2, 3, 4 }), ({}) }). Met functie *export_array()* kan dit array opgevraagd worden door andere objecten. Het array is van het type "mixed array" om het voorbeeld niet te complex te maken. De broncode voor deze entiteit is te vinden in figuur 3.5.

2 **Village green**
Deze entiteit bevat een globaal array ({ 'a', 'b', 'c' }), dat kan opgevraagd worden met functie *export_array()*. Ook dit array heeft elementen van het type mixed, voor dezelfde reden. Figuur 3.6 geeft de broncode voor deze entiteit.

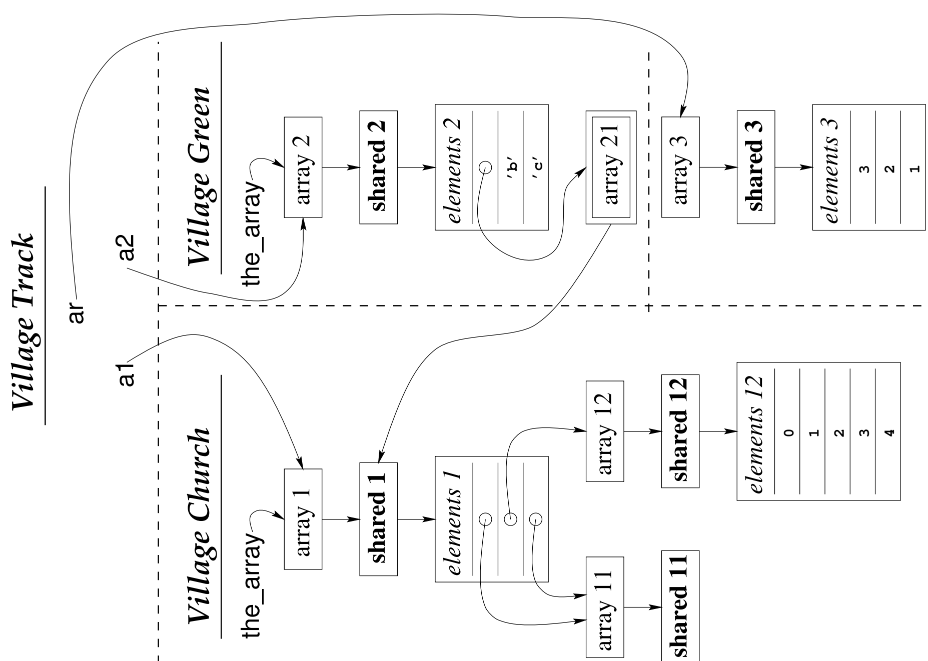
3 **Village track**
Hier zijn de acties ondergebracht die de import van arrays sturen. Uit de twee andere entiteiten worden de arrays opgevraagd, toekenningen worden uitgevoerd, en dan zal bij beëindiging van de uitvoering het copiëermechanisme in werking treden. Deze entiteit wordt gegeven in figuur 3.7. Merk op dat functie *doit()* voor de effectieve imports zorgt.

Stap 1: Initialisatie

Figuur 3.8 toont de situatie nadat de entiteiten geladen zijn. De creatie–functies zijn uitgevoerd, en dus hebben de entiteiten *Village church* en *Village green* hun arrays in eigendom.



Figuur 3.8: Ilustratie van het exporteren van arrays (stap 0)



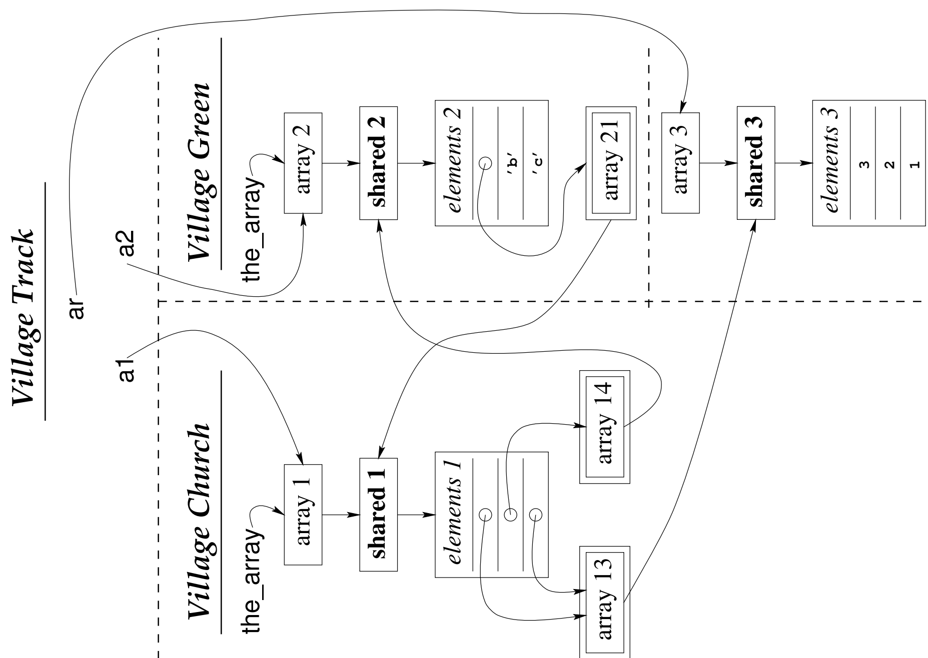
Figuur 3.9: Ilustratie van het exporteren van arrays (stap 1)

Stap 2: Eerste import

Lijnen 1 en 2 in figuur 3.7 halen de arrays uit de andere twee entiteiten op en kennen deze toe aan lokale variabelen. Deze arrays worden dus *niet* geïmporteerd in *Village track* omdat ze niet toegekend worden aan globale variabelen.

Lijn 3 initializeert een vlottend array (*ar*) voor de functie *doit()*.

Uiteindelijk wordt er dan een eerste import gedaan in lijn 4. Het betreft hier een onrechtstreekse import, aangezien een extern array (*a1*) wordt toegekend aan een element van een ander extern array (*a2*). Dit betekent dat array *a1* geïmporteerd wordt in de dataspace van *Village green*.



Figuur 3.10: Ilustratie van het exporteren van arrays (stap 2)

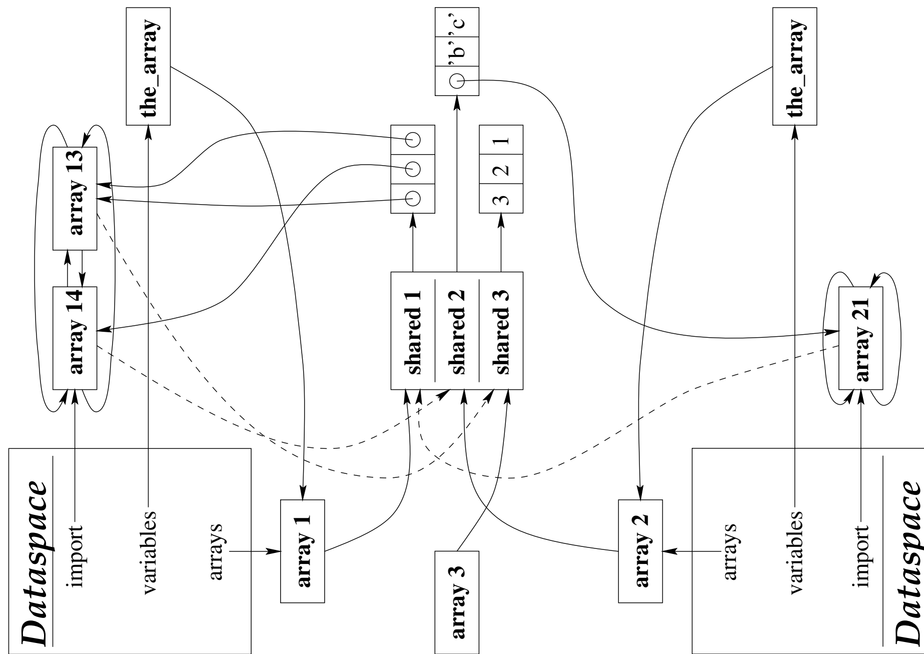
Stap 3: Verdere imports

Vervolgens wordt er in lijn 5 een toekenning gedaan van vlottend array *ar* aan het eerste[26] element van array *a1*. Dit is geen import van een array omdat *ar* vlottend was. Wat er zal gebeuren is dat de dataspace van entiteit *Village church* eigenaar wordt van het array.

Lijn 6 doet dan weer een onrechtstreekse import van array *a2* in dataspace *Village church*. Dit creëert een circulaire verwijzing tussen de arrays *a1* en *a2*. Zoals uit experimenten is gebleken, kan deze structuur pas opgekuisd worden bij een globale vernietiging van het geheugen (na swapping van de gehele objectwereld).

Tenslotte wordt in lijn 7 een tweede toekenning gedaan van *ar* aan een element van *a1*. Vermits *ar* eigendom is geworden van dataspace *Village church* in lijn 5, is dit geen import.

[26]LPC gebruikt een nummering van 0 tot N-1 voor arrays met N elementen.



Figuur 3.11: Ilustratie van het exporteren van arrays (interne representatie)

Het resultaat op het einde van functie *doit()* is te zien in figuur 3.10. De interne representatie van het voorbeeld nadat de arrays geïmporteerd zijn, is te zien in figuur 3.11.

Entry	Reference
Array 2	Import 1
Array 22	Copy–Values 1(1)1

Tabel 3.1: De unificatietabel

Stap 4: Uitvoering van het copiëermechanisme

Op het einde van de uitvoeringscyclus wordt het copiëermechanisme gestart, en zal allereerst de dataspace van *Village green* verwerkt worden in algoritme Import. In tabel 3.1 kan gevolgd worden hoe de unificatie van arrays verloopt.

Allereerst wordt het ene array dat eigendom is van de entiteit in de unificatietabel opgenomen. Vervolgens wordt de importlijst afgewerkt, die bij deze entiteit slechts één array bevat, namelijk *array 21.* Algoritme Import–Array wordt opgeroepen en het blijkt dat punt 2 *niet* van toepassing is voor dit array, en dus wordt naar punt 3 gegaan. Een nieuw gedeeld array wordt aangemaakt, en de elementen ervan worden gecopiëerd met behulp van algoritme Copy–Values.

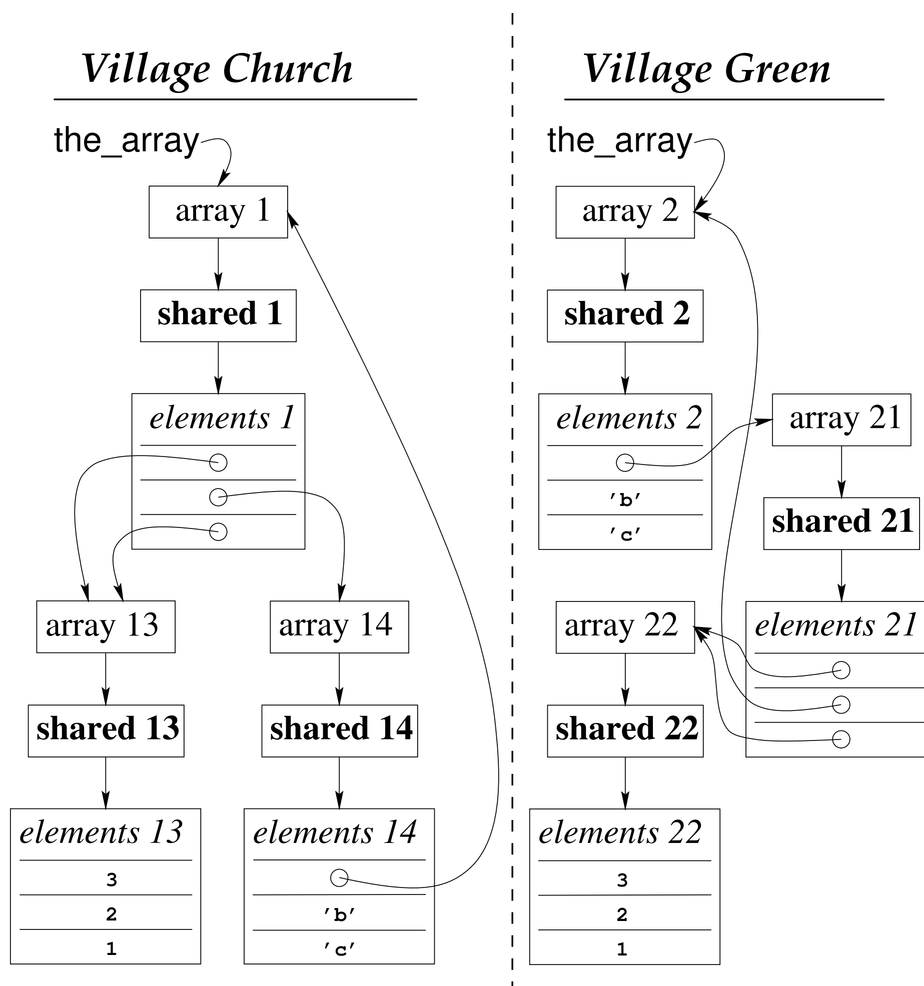
1 Het eerste element is *array 13.* Het gedeelde array van dit array werd nog niet verwerkt en dus wordt algoritme Import–Array opgeroepen. Vermits het array eigendom

is van een andere dataspace (namelijk *Village church*), zal het array geheel gecopiëerd worden. Geen van de elementen ervan zijn van het array of mapping type. Tenslotte wordt de nieuwe copie aan de unificatietabel toegevoegd.

2 Het tweede element is *array 14*. Voor dit array werd het gedeeld array reeds verwerkt, omdat blijkt dat *array 2* ook naar dit gedeeld array verwijst. Via de unificatietabel wordt een wijzer naar *array 2* bekomen, en het tweede element van *array 21* wordt vervangen door deze wijzer.

3 Het derde element tenslotte is weer *array 13*. Uiteraard wordt ontdekt dat dit array reeds verwerkt werd, en net als in het vorige geval zal de wijzer vervangen worden door de wijzer die uit de unificatietabel gehaald wordt.

De verwerking voor de dataspace *Village church* is analoog.



Figuur 3.12: Ilustratie van het exporteren van arrays (stap 3)

Stap 5: Na uitvoering van het copiëermechanisme

Ter vervollediging van dit voorbeeld geeft figuur 3.12 het resultaat na werking van het copiëermechanime. Beide entiteiten hebben nu een eigen copie van de datastructuur die volgt uit de uitvoering van *doit()* in entiteit *Village track*.

3.5.7 Een onopgelost probleem?

Felix Croes heeft nog een probleem ontdekt in de implementatie van het copiëeralgoritme, waar niet direct een oplossing voor gevonden werd. Het is namelijk zo dat indien een array X uit object A in object B geïmporteerd wordt, er een array Y voor wordt aangemaakt die naar het gedeeld array zal wijzen. Indien nu een array Z in B aan een element in Y wordt toegekend, is dit geen import omdat Y eigendom van B is.

Wanneer nu binnen dezelfde uitvoeringscyclus de verwijzing in B naar Y verwijderd wordt, blijkt het array Z uit B impliciet geïmporteerd te zijn in A, zonder dat het voorkomt in de importlijst van A.

Gelukkig werd op het allerlaatste moment toch een oplossing gevonden. Zoals uit voorgaande beschrijving van het probleem kan afgeleid worden, moet er een manier gevonden worden om enerzijds te weten dat het een toekenning betreft aan een geïmporteerd array, en anderzijds om te weten in welke dataspace het array zich bevindt waarvan geïmporteerd werd. Dit kan gedaan worden door in het beschrijvingsblok van een array een verwijzing toe te voegen naar de dataspace waaruit geïmporteerd wordt. Dan moet enkel bij toekenning aan een element van een array getest worden of deze verwijzing bestaat, en indien dat het geval is moet het array dat toegekend wordt in die dataspace expliciet geïmporteerd worden. Dit lost het bovenstaande probleem proper op.

Als een positief neveneffect van deze oplossing verzekert dit dat bij toekenning aan een geïmporteerd array *altijd* toegekend zal worden op basis van de dataspace waar het bronarray vandaan kwam[27].

[27] Het kan ondertussen reeds vernietigd zijn. Dit is geen probleem omdat het geïmporteerd array nog verwijst naar het gedeeld array waarin de toekenning was gebeurd.

Hoofdstuk 4

De LPC programmeeromgeving

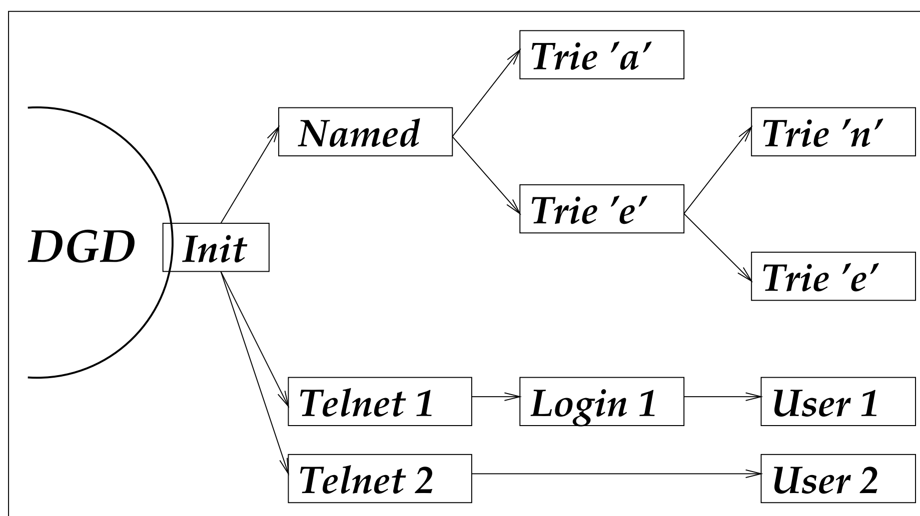
"It's never the technical stuff that gets you in trouble. It's the personalities and the politics."
(Gezegde bij programmeurs)

De programmeeromgeving waarmee de gebruiker in contact komt is geheel geschreven in LPC. Dit betekent voornamelijk dat deze omgeving zelf reeds een voorbeeld is van de mogelijkheden van deze programmeertaal en dat de broncode voor deze omgeving beschikbaar is voor alle gebruikers. Ook heeft dit als belangrijk voordeel dat aanpassingen eenvoudig zijn, en geen aanleiding geven tot veranderingen aan DGD. De keuze om de omgeving in LPC uit te werken is ook niet ongewoon, zeker niet in functie van de MUD–basis waarop deze thesis steunt. De spelomgevingen zijn namelijk ook stuk voor stuk geïmplementeerd in LPC.

Gezien het feit dat de broncode van de programmeeromgeving beschikbaar is voor alle gebruikers, is het zeker gewenst dat deze code duidelijk en algemeen is. Dit betekent dat het ontwerp zeker de object gerichte principes goed dient te volgen. Voldoende generalisatie leidt tot een duidelijk ontwerp, en laat toe op eenvoudige wijze aanpassingen uit te voeren. Toch kan het voorkomen dat een dergelijke generalisatie ongewenst is. Op pagina 43 werd beschreven dat er een kip–en–ei probleem bestaat bij de conversie van arbitraire namen naar bestandsnamen. Omdat de conversie op LPC niveau gebeurt, zijn er een aantal klassen nodig om deze functionaliteit te implementeren. De klassen die nodig zijn kunnen echter niet via de conversietabel bekomen worden, omdat dit wederom een kip–en–ei probleem veroorzaakt. Zodoende is het nodig deze namen in de statische tabel op te nemen. Deze tabel dient geminimaliseerd te worden, opdat het principe van de dynamische aanpasbaarheid geldig blijft. Hier zit dus een duidelijk dilemma: enerzijds is een zeer algemene implementatie gewenst, en anderzijds wordt een mininalisatie–principe nagestreefd. In dit geval wordt gekozen voor de minimalisatie van de statische tabel omdat dit de werking van de programmeeromgeving zeker ten goede komt. Ook is het nut van de klassen die in de statische tabel voorkomen erg beperkt.

Een bespreking van alle klassen en entiteiten die nodig zijn voor de programmeeromgeving is op basis van beperkingen op de grootte van deze thesis niet haalbaar. Er zal dan

ook slechts een deel van de klassen besproken worden. Het doel van deze bespreking is niet een handleiding te geven tot het systeem, maar eerder om aan te geven waarom en hoe bepaalde klassen werden ontworpen. Zoals zal blijken uit dit hoofdstuk, komen er aspecten aan bod in het ontwerp van klassen en entiteiten in LPC die tot nog toe niet zo algemeen in object gericht programmeren aanwezig zijn.



Figuur 4.1: DGD—LPC interface objecten

4.1 De basis van de LPC omgeving

Alvorens aan te geven hoe de interactie verloopt tussen gebruikers en de LPC omgeving, is het noodzakelijk uitleg te geven betreffende de werking en het ontwerp van de onderliggende code. Zoals reeds eerder werd vermeld, wordt de interface tussen de DGD server en de LPC omgeving gevormd door de driver entiteit. Ook bestaat er een rechtstreekse interactie tussen de server en de objecten die netwerkverbindingen beheren. Tijdens de opstartfase van de LPC omgeving (die uitgevoerd wordt door het oproepen van *initialize()* in de driver entiteit) moet de naamconversie code geladen worden, opdat zo snel mogelijk met de dynamische structuur gewerkt kan worden. Op figuur 4.1 wordt voorgesteld hoe men zich de object–structuur moet voorstellen voor wat de server—LPC interface betreft.

De driver entiteit creëert de “named” entiteit die de basis vormt van de naam conversie structuur. Zoals op de figuur gezien kan worden wordt er een hiërarchische structuur opgebouwd van “trie” objecten. Dit is een redelijk efficiënte manier om de conversietabel bij te houden en opzoeken makkelijk te maken. De keuze om een geheel specifieke implementatie te maken volgt uit de eis de statische tabel minimaal te houden. De statische conversietabel wordt in tabel refconvtable gegeven.

Verder worden er zodra de named entiteit geladen is, nog een aantal speciale entiteiten opgeladen om globale gegevens bij te houden zoals bijvoorbeeld de lijst van gebruikers,

Naam	Bestandsnaam
driver	/kernel/init
named	/kernel/named
object	/core/object
savable	/core/savable
trie	/core/trie

Tabel 4.1: De statische conversietabel

paswoorden, toegangsrechten, ... Deze entiteiten zijn de basis voor de rest van de gebruikersomgeving. Discussie betreffende het feit of dit entiteiten of instantiaties van klassen moeten zijn is mogelijk, doch in de huidige opzet is er zeker geen voldoende reden om instantiaties boven entiteiten te verkiezen.

Uiteraard zijn er nog andere klassen nodig om de omgeving te vervolledigen. Er moet bijvoorbeeld een klasse bestaan die de interactie vormt tussen de gebruiker en de bijhorende netwerkverbinding. De triviale oplossing zou in dit geval bestaan uit een klasse waarvan objecten worden gemaakt op het moment dat een netwerkverbinding aangegaan wordt. Dit object kan dan toegangsverificatie doen en verder de interactie verzorgen met de gebruiker. Deze methode is echter niet wenselijk voor verschillende redenen. Enerzijds is de toegangsverificatie geen onderdeel van de verdere interactie, en anderzijds is de creatie van een werksessie een gevolg van een succesvolle toegangscontrole. Het is dan ook niet wenselijk om deze twee aspecten met elkaar te vereenzelvigen. Vandaar dat er drie klassen gebruikt worden[1]. Allereerst de “telnet” klasse die de netwerkverbinding beheert, verder de “login” klasse die de toegangsverificatie zal verzorgen, en tenslotte de eigenlijke “user” klasse die de verdere interactie verzorgt tussen de gebruiker en de LPC omgeving.

4.2 De toegangsprocedure: een voorbeeld

Als voorbeeld van de werking van LPC is het goed eens te kijken wat er gebeurt indien een gebruiker een netwerkverbinding aangaat met de server. Om het voorbeeld eenvoudig te houden wordt er vanuit gegaan dat de verbinding via een gewone telnet toepassing gebeurt[2]. In bijlage B is de broncode voor deze objecten te vinden. Figuur 4.2 geeft de dialoog weer zoals deze op het scherm van de gebruiker zal verschijnen.

1 **Ontvangst van de verbinding in de server**
Zodra de gebruiker telnet gestart heeft, zal via het netwerk een poging gedaan worden een verbinding op te zetten tussen de gebruiker en de server. De server zal de aanvraag tot verbinding detecteren, en voor ontvangst zorgen.

[1] Er zijn wel degelijk drie klassen nodig en niet twee, omdat het object dat met een verbinding geassocieërd wordt later niet kan vervangen worden door een ander object.

[2] Het programmeersysteem ondersteunt ook het HTTP protocol dat voor World Wide Web gebruikt wordt. Dit is een eenvoudig prototype van een grafische gebruikersinterface.

WELCOME TO HOPE...

RUNNING DGD 1.0.9.1T.
PLEASE ENTER YOUR NAME: **aedil**
PLEASE ENTER YOUR PASSWORD:

Figuur 4.2: De toegangsdialoog

2 **Creatie van een telnet object**
Omdat de verwerking van netwerkverbindingen op LPC niveau gebeurt, zal de server aan de driver entiteit opdracht geven een telnet object aan te maken, door middel van een oproep van *telnet_connect()*. Dit laat toe in LPC code extra functionaliteit toe te voegen zoals bijvoorbeeld een keuze tussen meerdere telnet klassen.

3 **Creatie van een login object**
De driver entiteit zal *open()* oproepen in het telnet object dat als waarde werd teruggegeven bij de oproep van *telnet_connect()*. Deze functie kan allerlei initializatie doen, en een login object zal aangemaakt worden. In dit login object zal ook enige initializatie uigevoerd worden, en zal via een oproep van *redirect* in het telnet object aangegeven worden dat invoer van de gebruiker herleid moet worden naar de functie die de toegangsverificatie zal aanvangen, namelijk *get_name()*. Tenslotte wordt uitvoer naar de gebruiker verzonden, om te vragen een gebruikersnaam op te geven.

4 **Eerste stap van de toegangsverificatie**
Zodra de gebruiker invoer geeft[3] zal DGD deze tekst van de verbinding inlezen en functie *receive_message()* oproepen in het bijhorende telnet object. De invoerlijn zal als parameter meegegeven worden.

Het betreffende telnet object zal deze invoer rechtstreeks doorgeven aan het login object, door *get_name()* op te roepen. Hier wordt er een verificatie gedaan om te zien of de naam wel aanvaardbaar is, en of de gebruiker bestaat. Indien deze verificatie niet succesvol is, wordt de verbinding verbroken, en zullen zowel het telnet object als het login object vernietigd worden.

Bij een succesvolle verificatie van de gebruikersnaam zal ingesteld worden dat verdere invoer naar *get_pass()* herleid moet worden.

5 **Tweede stap van de gebruikersverificatie**
Tenslotte zal bij de volgende invoerlijn, **get_pass()** opgeroepen worden. Deze functie zal vervolgens aan de "password" entiteit vragen of de gegeven invoer juist is als wachtwoord voor deze gebruiker, en indien dit niet het geval is, wordt de connectie afgesloten, en zullen beide objecten everzeer vernietigd worden.

[3]De telnet verbinding zoals deze door DGD verzorgd wordt, werkt lijngestuurd. Dit betekent dat de telnet toepassing van de gebruiker een invoerlijn slechts zal doorzenden op het moment dat de gebruiker hiertoe opdracht geeft (meestal door middel van het indrukken van de "Return" of "Enter" toets).

Bij een succesvolle verificatie van het wachtwoord, zal een user object aangemaakt worden, en het telnet object zal de verwijzing naar het login object vervangen door een verwijzing naar het user object. Dit is een legale aktie omdat beide klassen via inheritance een specialisatie zijn van de “connection” klasse.

6 **Einde van de toegangsprocedure**
Wanneer de toegangsprocedure beëindigd is (concreet betekent dit dat het login object zichzelf heeft vernietigd), verloopt de interactie met de gebruiker dus via de volgende weg: telnet toepassing → DGD server → telnet object → user object.

4.3 De dialoog met de gebruiker

Aangezien er met een programmeeromgeving gewerkt wordt, is het uiteraard niet zo dat er veel aandacht besteed werd aan de visuele aspecten. De nadruk ligt op de mogelijkheid programmeerexperimenten uit te voeren, en deze te analyseren voor wat betreft de werking en voornamelijk ook voor de samenwerking met andere programmacode. In die zin is het prototype dat ontwikkeld werd niet zo erg gebruiksvriendelijk. Om uitbreidingen en verbeteringen in de toekomst mogelijk te maken, werd ondersteunende code geschreven. Het is dan ook zo dat de huidige interface eerder bedoeld is om het prototype te kunnen demonstreren dan voor een meer serieus gebruik. Er staat echter niets in de weg van een uitbreiding van de gebruikersinterface in een later stadium.

Alvorens in te gaan op een aantal aspecten van de gebruikersinterface is het best een interpretatie–probleem uit de wereld te helpen. Het is zo dat van een object gericht systeem uiteraard verwacht wordt dat de interface ook object gericht is. Een procedurale interface wordt als voorbijgestreefd beschouwd. Dit is inderdaad een juiste opvatting. Het is echter noodzakelijk een goed onderscheid te maken tussen een procedurale interface en een interface op basis van de menselijke taal. Om een voorbeeld te nemen uit de MUD wereld:

insert key in keyhole

Dit kan beschouwd worden als een proceduraal commando. Functionaliteit “insert” wordt uitgevoerd voor “key” en “keyhole”. In die zin is dit dus zeker niet goed voor een object gericht systeem. Anderzijds kan dit ook verklaard worden als een commando dat het dichtst bij de menselijke taal staat. De object gerichte variant lijkt heel wat kunstmatiger:

keyhole -> insert(key)

Uit object gericht oogpunt is dit beter, doch indien een gebruiker dergelijke commando’s zelf moet intypen kan dit tot verwarring en frustratie leiden. Het is niet de bedoeling een gebruiker te verplichten om object gericht te werken indien dit minder praktisch blijkt te zijn. Al bij al zijn er zowel voordelen als nadelen verbonden aan de drie verschillende interfaces. De keuze is dan ook erg subjectief. Voor een grafische gebruikersinterface is een object gerichte aanpak zeer intuïtief en handig in het gebruik. Een beperkt prototype van

een dergelijke interface is aanwezig in de programmeeromgeving van deze thesis op basis van World Wide Web. Dit is enerzijds een experimentele interface om de werking van programmacode te kunnen analyseren, en anderzijds een voorbeeld van de mogelijkheden van de omgeving. De interface bestaat uitsluitend uit een hele reeks klassen in LPC.

De dialoog tussen de programmeeromgeving en de gebruiker bestaat dus effectief uit twee onderdelen. Enerzijds de tekstuele dialoog en anderzijds de dialoog via een World Wide Web toepassing.

4.3.1 De tekstuele dialoog

De tekstuele dialoog tussen de gebruiker en de omgeving verloopt via de normale weg van de netwerkverbinding die reeds vermeld werd bij de toegangscontrole. De gebruiker typt commando's in zijn telnet toepassing en deze worden lijn per lijn doorgestuurd naar de DGD server. Via het telnet object en het user object die met de verbinding geassociëerd zijn komen deze commando's terecht bij een commando analyse. Hier wordt bepaald wat er juist gedaan moet worden met de invoer van de gebruiker. Twee klassen zijn er hier van belang:

1 **De "virtual room" klasse**
Deze klasse stelt een aantal specifieke werkcommando's ter beschikking aan de gebruiker. Deze commando's bieden een interface aan die poogt op een gebruiksvriendelijke manier functionaliteit te geven. Deze interface is zoveel mogelijk gebaseerd op de menselijke taal, en bevat commando's om bestandsmanipulaties uit te voeren, bestanden te lezen, berichten naar andere gebruikers te sturen, ...

2 **De "actor" klasse**
Om toe te laten dat ook andere klassen in staat zijn commando's uit te voeren is de commando verwerking ondergebracht in een specifieke klasse. Deze klasse vormt een soort versmelting van de interface volgens object gerichte principes en die volgens menselijke taal. Alle commando's die door een gebruiker ingetypt worden komen in deze klasse terecht. Hier wordt beslist wat er met het commando gedaan moet worden. Concreet betekent dit voornamelijk dat er beslist wordt tussen:

- Een functie oproepen in een object. Dit is de object gerichte interface naar de gebruiker toe omdat geëist wordt dat de gebruiker aangeeft in welk doelobject een bepaalde functie moet opgeroepen worden, met parameters die tevens onderdeel zijn van het commando. Bovenstaand voorbeeld van object gericht werken geldt hier.
- Een speciaal commando dat in de "virtual room" gedefiniëerd is wordt uitgevoerd door het commando door te geven aan de betreffende kamer. Elke gebruiker heeft een eigen kamer die de basis vormt van zijn eigendom binnen de objectwereld. Dit is een interface gebaseerd op menselijke taal.

- Tenslotte kan de invoer van de gebruiker een commando zijn dat de toestand van de commando analyse aanpast. Denk hierbij aan verkorte namen voor objecten, een object aangeven waarin alle volgende functies opgeroepen moeten worden (zodat niet steeds hetzelfde doel moet opgegeven worden). Ook hier is de interface op basis van menselijke taal.

Op deze manier kan de gebruiker in interactie treden met de LPC objecten die zich in de abstracte virtuele wereld bevinden.

4.3.2 De WWW dialoog (fakultatief)

De WWW dialoog is enerzijds minder complex doordat er uitsluitend via keuzen in de grafische voorstellingbeslist kan worden welke akties uitgevoerd moeten worden. Anderzijds is de dialoog echter complexer doordat er een hele verzameling klassen nodig is om de communicatie mogelijk te maken. Zonder dieper op deze materie in te gaan is het nuttig te vermelden dat er een klasse ontwikkeld werd om de HTTP[4] dialoog uit te voeren, en verder nog een hele verzameling klassen die bovenop dit geheel de HTML taal[5] implementeren.

Volgende informatie is via deze interface beschikbaar voor de gebruiker:

- Hulpdocumenten die uitleg geven over primitieve functies, algemene commando's, ...
- Structuurbeschrijvingen van klassen en entiteiten, met onder meer informatie betreffende welke functies vanuit andere objecten opgeroepen kunnen worden, en informatie betreffende inheritance.
- Toestandsinformatie van objecten in de wereld.
- Informatie betreffende de objectverzamelingen die eigendom zijn van een bepaalde gebruiker, met verwijzingen naar meer specifieke informatie over die klassen en entiteiten, en afgeleide objecten, zoals hierboven vermeld wordt.

4.4 Een voorbeeld

Als voorbeeld van de programmeeromgeving wordt in dit deel een kleine impressie gegeven van de werking ervan. Om de complexiteit te beperken wordt een vrij kunstmatig voorbeeld genomen, namelijk een object gerichte implementatie van het zeefalgoritme van Aristothenes om priemgetallen te genereren. Neem aan dat de gebruiker reeds tot het systeem toegelaten werd. Commando's van de gebruiker worden aangegeven met het prefix

[4]HTTP staat voor HyperText Transfer Protocol, en vormt de basis voor de communicatie tussen World Wide Web toepassingen.

[5]HTML staat voor HyperText Markup Language, en definiëert een universele taal om documenten voor te stellen.

"HOPE¿ ", invoer in de tekstverwerker krijgt het prefix ": ", en uitvoer van het systeem wordt gegeven zonder prefix.

De basis van het algoritme is een nummergenerator, een uitvoer entiteit en een filter. Steeds wanneer er een priemgetal gevonden wordt, wordt er een nieuwe filter gemaakt met dat getal, en deze wordt toegevoegd aan het einde van de reeds bestaande lijst, net voor de uitvoer entiteit. Zowel de generator klasse als de uitvoer entiteit hangen via inheritance af van de filter, om toewijzingen van objecten de juiste semantiek te geven.

In de eerste faze wordt geïllustreerd hoe een object verzameling gekozen wordt. Daarna wordt één van de klassen ingeladen in de tekstverwerker om een kleine aanpassing uit te voeren. Zoals kan gezien worden geeft de tekstverwerker de effectieve naam van de entiteit.

```
You are in your workroom, full of file cabinets and long lists  of
notes.  Objects are piled up in a far corner and a fresh notebook,
bare of notes for now, is on the desk before you.

HOPE> open cabinet scratch
You open the file cabinet, flipping through the index sheets, and
notice:
  sieve counter
  sieve output
  sieve stage

HOPE> edit sieve output
"/players/aedil/scratch/output.c" 12 lines, 230 characters
: 1z
entity "sieve output" {
    inherit "stage";

    void out(int i) {
        object      ob;

        write(i + " is prime.\n");

        ob = "stage"->clone(i, this_object());
        previous_object()->change_output(ob);
    }
}
: 1,$s/"stage"/"sieve stage"/g
        ob = "sieve stage"->clone(i, this_object());
: wq
"/players/aedil/scratch/output.c" 12 lines, 242 characters

HOPE>
```

Vervolgens gaat de gebruiker zijn zeefalgoritme in werken zetten. Dit gebeurt door een “sieve counter” object te laden. Dit object krijgt als parameter bij de creatie een getal dat het einde van de reeks getallen aangeeft waarbinnen er naar priemgetallen gezocht moet worden. Om het systeem niet volledig aan het werk te zetten in één lange iteratie wordt er in deelintervallen gewerkt. Zoals blijkt kan de gebruiker ondertussen andere dingen doen.

```
HOPE> sieve counter->clone(33);
Beginning of execution...
Sieve chunk from 1 to 11.
Primes are: 2 3 5 7 11

HOPE> tell leda Hey, this thingy even works!
You tell Leda: Hey, this thingy even works!
Sieve chunk from 12 to 22.
Primes are: 13 17 19
Leda tells you: Hm, goodie.  By the way, is 31 prime?

HOPE> rdo leda smiles and nods.
You display on Leda's screen: Aedil smiles and nods.
Sieve chunk from 23 to 33.
Primes are: 23 29 31
End of execution...
```

4.5 Ontwerpaspecten

Eén van de belangrijke aspecten in een omgeving die op MUDs gebaseerd is, is de continuïteit van de omgeving. Het is niet de bedoeling dat de omgeving ooit ophoudt met bestaan. Uiteraard is dit een ideaal dat nooit bereikt kan worden, doch in principe moet alles zodanig ontworpen worden dat in een oneindig lang draaiend systeem geen problemen ontstaan. Dit betekent voornamelijk dat er eisen gesteld worden aan de werking van klassen en entiteiten. Zo is het nodig dat objecten die niet meer nodig zijn, vernietigd worden om plaats te maken voor nieuwe objecten. Ook dient men er zorg voor te dragen dat objecten op een degelijke manier ontworpen worden om te verzekeren dat latere aanpassingen in klassen geen problemen veroorzaken (voor zover de semantiek bewaard blijft). Er zijn ook efficiëntie aspecten verbonden met een systeem dat zonder onderbrekingen moet blijven werken, doch deze zijn minder van belang.

De theorie van het object gericht programmeren neemt herbruikbaarheid als een redelijk belangrijk aspect van het paradigma. In praktijk blijkt dit echter slechts in heel beperkte mate toegepast te worden. Omgevingen veranderen, en de evolutie van programmatuur laat eigenlijk niet toe herbruikbaarheid goed toe te passen. Vermits continuïteit meer en meer nagestreefd wordt in systemen komt dit aspect meer in de belangstelling. Niet zozeer in de strikte zin van een herbruiken van klassen, maar ook in de meer algemene betekenis

van het herbruiken van gehele objecten en entiteiten. Denk namelijk maar aan een groot systeem dat voor vluchtreservaties gebruikt wordt bij de luchtvaartmaatschappij. Op een bepaald moment kan men beslissen de gebruikersinterface te vervangen door een andere interface. Het systeem wordt ononderbreekbaar verondersteld, en dus moet men kunnen steunen op het feit dat de vervanging van één element redelijk transparant kan gebeuren.

In deze zin is het nut van de programmeeromgeving die werd uitgewerkt voor deze thesis een mogelijke basis om de aspecten van een altijd draaiend systeem te bestuderen, en mensen ermee in contact te brengen.

Hoofdstuk 5

MUDs

“Any sufficiently advanced technology is indistinguishable from magic.”
(Clarke’s law)

MUDs zijn een erg populaire netwerktoepassing, voornamelijk omdat spelletjes nu eenmaal erg veel aandacht trekken. Er bestaan evenwel heel wat misvattingen over dit fenomeen. Zo werd lange tijd gedacht dat MUDs een erg grote belasting vormen op het internationale netwerk. Gelukkig werd aangetoond dat dit wel degelijk een foute visie was. Een normale MUD zal zelden meer belasting op het netwerk geven dan bijvoorbeeld een gewone werksessie.

MUDs zijn echter niet enkel spelletjes. Reeds vroeger in deze thesis werd melding gemaakt van een WWW server die op basis van een MUD werkt, en ook deze thesis zelf is een voorbeeld van een meer serieuze toepassing van deze “technologie”. Andere voorbeelden zijn de Astro–VR en Jupiter projecten die beschreven worden in [CN93]. Het betreft hier omgevingen die speciaal gemaakt werden om informatie–uitwisseling tussen astronomen in een mogelijk meer intuïtieve vorm aan te bieden.

Een zeer belangrijk aspect van MUDs is het gevoel in een virtuele wereld rond te wandelen. Misschien lijkt het raar dat van een virtuele wereld wordt gesproken bij een toepassing die geheel met tekst werkt. Toch blijkt uit studies[1] dat spelers van MUDs de spelwereld als een virtuele realiteit aanvaarden, zelfs zonder dat er enige grafische aspecten aanwezig zijn. Dit blijkt te maken te hebben met de navigatorische aspecten, en uiteraard de menselijke fantasie. Het gaat zover dat een redelijke groep onderzoekers vindt dat grafische effecten eerder een aandachtstrekker zijn in plaats van een aspect van virtuele realiteiten. Dit is wel een erg drastische opvatting, en moet waarschijnlijk met een korrel zout genomen worden.

Het volgende fragment is een weergave van het leven in de virtuele wereld van een MUD. Een speler genaamd Lexus werd tijdens een spelsessie gevolgd, en de hele sessie werd rechtstreeks van haar scherm naar een bestand geschreven. Tijdens dit eerste fragment gaat ze op weg naar de “Adventurer’s Guild” om daar de grafitti en het prikbord te lezen.

[1] Jolanda Tromp heeft hier enige studies aan gewijd, doch tot op heden is geen gepubliceerde referentie bevonden.

Xyppe zegt ondertussen iets op een globale communicatielijn[2]. Andere spelers komen en gaan, en op het einde is te zien hoe Radja een persoonlijk bericht stuurt naar Lexus, en antwoord krijgt.

```
A long road going through the village. There are stairs
going down. The road continues to the west.
To the north is the shop, and to the
south is the adventurer’s guild. The road runs towards
the shore to the east.
      Obvious exits: east, north, south, west and down.
Xyppe is on the party line: Hey Radja, that bastard sword got me booted
      out of Mauve for not using a worthy weapon...it’s not really a
      sword...
> s
You have entered the Adventurer’s Guild.
This is the main gathering point for adventurers all over Igor.
There are guild masters present who will help you gain your
new level or stats when you are ready. Use the command ‘cost’
to know, and ‘advance [level, str, con, int, dex]’ when you do.
To see what perilous quests you must undertake, use ‘list’.
There is a strange, shimmering blue light in the southern doorway.
From here the role-playing board is up and the LPmud board is east.
      Obvious exits: east, north, south, up, west and down.
Morningwood and Anjelique are here.
A graffiti plank, the Adventurer’s Cat Board, a word box and a book on
a chain are here.
The Adventurer’s Cat Board licks you.
> read plank
This is what’s scribbled on the plank:
Violetwing kissar i din g}rd.
The Mauve Wave shall sweep over Igor!!!!
Whee.
Agrivar is my honeybunny!!!!
Whee this!
only 78 days to go....    *bounce*
Hej, jag e ny h{r....
To make some graffiti of your own, use ’scribble Kilroy was here!’
This plank is Polgara’s creation, coded by Zell.
Anjelique crystallizes and shatters into 1000 pieces.
Helmut arrives.
> exa box
```

[2]Deze communicatielijnen zijn een belangrijk sociaal aspect in MUDs. Spelers kunnen lid worden van een club en dan met andere leden communiceren via deze lijnen.

```
You look at the small box and a hand pops out giving you a note.
Word box gives a note to Lexus.
NB You can 'suggest' other words for the day.
>  read note
The paper has written on it in bold red (non puce) letters
The amazing fabulous and truly exciting word for today is 'fish'.
Please use this word in conversation at least once or suffer severe mental
trauma... One of the wizards might talk to you!
This word is brought to you by Igor's stupidity department.
PS Dont litter.
Lexus reads her word for the day note.
Helmut leaves north.
> exa tail
B
N 86: Hi /10/ Lyssa <14> Thu Apr 27 08:27:23
N 87: Gwen /4/ Kador <20> Thu Apr 27 10:50:22
N 88: Re: Kad /4/ Ysol <19> Thu Apr 27 13:06:28
N 89: Ysol /4/ Kador <20> Thu Apr 27 14:13:53
N 90: Blijn enzo /6/ Doctor <22> Thu Apr 27 17:14:01
N 91: Typos /4/ Doctor <22> Thu Apr 27 17:15:17
N 92: Re: Typos /3/ Carlsson <31> Thu Apr 27 18:07:23
N 93: Drugs! /2/ Takhisis <12> Thu Apr 27 18:39:00
Emeraude arrives.
> 86
[86] Hi /10/ Lyssa <14> Thu Apr 27 08:27:23

One person can have an idea,
Can see the old in a whole new way,
CAn tell a story, sing a song,
One person can reach out their hand
and be a friend...
One person carries the greatest power of all....
To love.

Be excellant to eachother.   *smile*

Radja tells you: Byebey, see you later.  Be well.
> tell radja Byebye!!!
You tell Radja: Byebye!!!
Your loverose tells you: Radja leaves the game.
Radja left the game.
```

Natuurlijk is het spelelement zeker nog aanwezig bij een MUD. Het volgende voorbeeld

laat dit duidelijk blijken, daar Lexus een gevecht aangaat met een krokodil. Ze maakt hierbij gebruik van magische spreuken en zelfs een echte magische cirkel. Belangrijk tijdens een gevecht is zeker de eigen fysische conditie en dus wordt die ook in het oog gehouden. Merk op dat het commando "sc" in feite een alias is voor "score". Dergelijke aliases zijn mogelijk met behulp van speciale objecten die een speler bij zich kan dragen.

```
> draw circle
With a small blade, you slice open your palm, letting blood flow freely.
Kneeling on the ground, you place your palm face down and
draw a circle. Within the circle, you draw a triangle.
You mutter: By the Man, the Dragon, and the God.
The fresh blood of the magic circle glistens.
For a moment, there is a pause and a strange silence.
You grazed Crocodile.
Crocodile missed you.
HP: 102 (234) SP: 13 (258)
You missed Crocodile.
Crocodile hit you hard.
HP: 96 (234) SP: 15 (258)
You hit Crocodile.
Crocodile hit you.
HP: 93 (234) SP: 15 (258)
> eat steak
It was VERY Spicy and delicious!
> You hit Crocodile hard.
Crocodile hit you hard.
HP: 117 (234) SP: 45 (258)
> sc
You are Lexus is swinging from the stars. (In Aedil's eyes) (good)
Level: 20
Hit points: 117 (234)   Spell points: 45 (258)
Experience: 1043336     Gold: 10104
Strength: 24            Magic: 27
Armed combat: 27        Constitution: 24
You are satiated.
Wimpy mode.
age:   34 days 10 hours 7 minutes 54 seconds.
> fireball
You cast a fireball.
You hit Crocodile very hard.
Crocodile died.
You killed Crocodile.
```

Tenslotte nog één conversatie uit vele die de auteur had met Dworkin, en zoals ge-

woonlijk, een gezonde en leerrijke discussie. In feite is een groot deel van deze thesis tot stand gekomen met dergelijke gesprekken die allemaal plaats vonden in MUDs. De informatie–uitwisseling die op deze manier mogelijk is, blijkt erg constructief te zijn.

```
You say: Well, I'm implementing two kinds of arrays: local arrays and
         exported arrays.
Dworkin says: But how do you differentiate between them?
Dworkin says: I don't think you can in all cases.
You say: Exported arrays can only result from passing them as arguments
         in a function call, and then assigning them to a global var or
         to an element of a global var.
Dworkin says: No, there are other ways.
You say: Oh?
Dworkin says: For instance, export an array that refers to other arrays.
Dworkin says: Then remove it from the original.
Dworkin says: Then all the arrays contained in this array are still in the
              old object.
You say: Well yes, but those arrays are then local to the old object and
         exported to the new object.  They might change owner eventually.
Dworkin says: But the top array is no longer referenced in the old object,
              so it is a local one.
You say: Nope, it still is an import in the new object, even when the
         original is gone.  Otherwise you get a definite mess.
```

Uit oogpunt van de volledigheid moet opgemerkt worden dat Dworkin ook een array importatie algoritme geïmplementeerd heeft op basis van de discussies. De conceptuele ondergrond is erg gelijk aan de oplossing die in deze thesis werd uitgewerkt. In dat opzicht heeft de ontwikkeling van deze thesis bij kunnen dragen tot de ontwikkeling van DGD, door aan te tonen dat de oplossing van de wijzertype anomalie zeker mogelijk is.

Hoofdstuk 6

Besluit

"Everything always takes twice as long and costs four times as much as you planned."
(Axioma voor programmeurs)

Zoals uit voorgaande hoofdstukken blijkt is het grootste werk in deze thesis te vinden in de studie en uitwerking van aanpassingen aan de LPC taal en de DGD server. Toen de thesis nog in het ontwerpstadium was, werd besloten om de aanpassingen aan DGD op een andere manier uit te voeren. Overleg met Felix Croes wees uit dat hij van plan was het array copiëermechanisme te implementeren, en zodoende werd besloten om in plaats van deze aanpassing een vorm van operator–herdefinitie in te voeren in LPC. Deze beslissing werd enerzijds genomen ter compensatie van het werk dat Felix Croes overnam, en anderzijds omdat dergelijke herdefinities erg nuttig kunnen zijn.

Uiteindelijk bleek echter dat Felix onvoldoende tijd kon besteden aan verdere ontwikkeling van DGD en daarom was het noodzakelijk het ontwerp te herzien. De implementatie van het copiëermechanisme als oplossing van de erg storende wijzer–type anomalie is een cruciaal deel van deze thesis omdat het de grootste fout uit LPC verwijdert. Reeds vanaf het begin van het ontwerp werd besloten dat het gedrag van wijzer–types fout was en onaanvaardbaar was voor een programmeertaal die gebruikt zou worden voor educatieve doeleinden. Ook is het gedrag duidelijk tegen object gerichtheid, en dat is ook een belangrijk punt geweest in de beslissing.

6.1 Het werk

De studie van de problematiek die aan de grondslag ligt van deze thesis was een tijdrovend werk wegens het gebrek aan literatuur. Er zijn wel een aantal officiëuze publicaties te vinden betreffende MUDs, doch deze behandelen zelden aspecten die bruikbaar zijn in deze thesis. Het enige echt relevante artikel is [Bru94]. Hier wordt besproken hoe MUDs gebruikt kunnen worden als een omgeving voor een leerproces gebaseerd op samenwerking. Dit is duidelijk een belangrijk aspect voor deze thesis zoals uit de inleiding reeds blijkt.

De implementatie van de thesis bestaat uit twee delen: de aanpassingen aan DGD en de implementatie van de LPC omgeving. Alvorens deze twee onderdelen nader te bekijken,

is het best even het doel van de thesis terug in herinnering te brengen:

> *De ontwikkeling van een object gericht programmeersysteem voor educatieve doeleinden.*

Dit lijkt impliciet aan te geven dat het grootste werk bestaat uit de ontwikkeling van de eigenlijke omgeving. Dit was ook de veronderstelling bij aanvang van de thesis. De praktijk blijkt echter uit te wijzen dat het grootste werk te vinden was in de aanpassingen van de programmeertaal (wat in praktijk neerkomt op aanpassingen aan DGD). Dit blijkt misschien niet direct uit het aantal lijnen programmacode, want al bij al werden slechts ongeveer 2900 lijnen C–code geschreven voor deze thesis, maar het voorbereidende werk en het testen heeft ook heel wat werk in beslag genomen. De complexiteit van aanpassingen zoals de naamgeving van klassen en entiteiten, en het oplossen van de wijzer–type anomalie kan zeker niet onderschat worden. Ettelijke dagen werk werden besteed aan de studie van de impact die een dergelijke aanpassing zou hebben, om een beeld te kunnen vormen van wat er juist allemaal verandert moest worden. Een ogenschijnlijk kleine aanpassing kan in een bestaand programma zoals DGD verstrekkende gevolgen hebben.

In dit opzicht kan men stellen dat de originele doelstellingen een natuurlijke verschuiving hebben ondergaan naarmate de thesis evolueerde. Waar in het begin de nadruk eigenlijk op de programmeeromgeving zelf lag, werd dit later meer genuanceerd en kwam de nadruk voornamelijk te liggen op de onderliggende code die de omgeving mogelijk moest maken. Dit werd reeds in de inleiding onder ogen gezien, en daarom is het ook reeds vanaf het begin de bedoeling geweest een prototype te maken van de omgeving. Met de verbeteringen die aan LPC werden uitgevoerd is het mogelijk een professioneel systeem uit te bouwen in plaats van het prototype dat nu bestaat.

Zoals reeds vermeld werd is in tegenstelling tot de oorspronkelijke planning al het implementatiewerk door de auteur zelf verricht. Dit heeft tot gevolg dat een aantal aanpassingen niet konden worden uitgevoerd, doch het lijkt duidelijk dat de keuze tussen een oplossing voor een anomalie en een extra functionaliteit helemaal niet moeilijk is. Extra functionaliteit is een luxe, terwijl een oplossing voor een anomalie tot een noodzakelijkheid verheven moet worden. Er is nooit een geldige verantwoording voor het verkiezen van luxe boven noden.

6.2 De doelstellingen: een terugblik

Dan rijst natuurlijk de vraag: “Werden de doelstellingen verwezelijkt?” Hier kan geen éénsluitend antwoord op gegeven worden.

- **Ja**
  De doelstellingen werden verwezelijkt. Er is een programmeeromgeving die werkt op de aangepaste versie van DGD die de opgekuiste versie van LPC implementeerd. Deze omgeving is een prototype, doch werkt voldoende om aan te tonen dat het LPC dialect goed geïmplementeerd is.

- **Neen**
  De doelstellingen spreken wel over een prototype, maar een prototype kan erg beperkt zijn, of erg uitgebreid. De grens is moeilijk te trekken en zodoende is een project als dit nooit beëindigd te noemen. Er blijft altijd wel iets dat nog uitgewerkt zou kunnen worden. Was dit niet het geval dan zou men eigenlijk zelfs van een mislukking kunnen spreken omdat het simpelweg onmogelijk is de perfecte werkomgeving te creëren. Gebrek aan toekomstmogelijkheden zou tot het besluit moeten leiden dat er ergens een beperkende factor ingevoerd werd die onnodig is.

In terugblik op de doelstellingen die in de inleiding uiteengezet werden moet ook opgemerkt worden dat enkele grenzen werden doorbroken. Pas laat in de ontwikkeling van de thesis werd besloten een World Wide Web interface toe te voegen. De redenen hiervoor zijn niet geheel wetenschappelijk te verantwoorden. Enerzijds leek het dadelijk een toepassing die zeer goed gebruikt kan worden ter demonstratie van het geleverde werk[1], en anderzijds was het gewoon een uitdaging om eens te proberen.

6.3 De toekomst. . .

Vermits slechts een prototype werd ontwikkeld is er nog heel wat ruimte voor werk in de toekomst. Dit werk kan in twee soorten opgesplitst worden. Allereerst zijn er nog heel wat mogelijkheden die uitgewerkt kunnen worden op de LPC implementatie die in deze thesis uitgewerkt werd. Denk daarbij aan een meer professionele werkomgeving, klassebibliotheken, hulpprogrammas, . . . Ook is er nog een hele wereld van MUD–gebaseerde toepassingen die eigenlijk zeer nuttig kunnen blijken in een educatieve omgeving. Voornamelijk de mogelijkheden voor communicatie tussen gebruikers is heel belangrijk. Met de huidige opkomst van multimedia en gedistribueerd onderwijs (bijvoorbeeld teleconferenties) is een server–gebaseerd systeem zeker een bron van mogelijkheden.

Verder zijn er nog heel wat mogelijkheden voor wat betreft de implementatie van de programmeertaal LPC. Zoals uit deze thesis duidelijk blijkt zijn alle implementaties nog steeds gebaseerd op sequentiële uitvoering. Een zogenaamd multi–threaded[2] systeem is zeker iets dat in de toekomst een belangrijke stap voorwaards zou zijn. Aspecten zoals consistentie komen dan aan de orde, doch het is zeker mogelijk. Onderzoek in de toekomst zou zich hier zeker op kunnen concentreren omdat het enerzijds een logische uitbreiding is, en anderzijds laat dit toe het fenomeen MUD (meer specifiek MUD gebaseerd op LPC) te gebruiken in andere onderzoeksgebieden. De interne vertolker van DGD is in feite een simulator voor een stackmachine met een erg beperkt instructieset. Ook op dit vlak is onderzoek mogelijk.

[1] Met praktische toepassingen. Er is interesse voor deze implementatie van het populaire hypertext systeem vanwege een commerciële netwerk organizatie in Zweden.

[2] Deze vakterm is heel moeilijk te vertalen. Het kan beschreven worden als een systeem met “meervoudige simultane uitvoering”, maar dit dekt de lading toch niet volledig.

6.4 Eindconclusie

Tijdens de afgelopen 10 maanden werden heel veel uren besteed aan de studie en implementatie van deze thesis en dit is veelal een zeer leerrijke tijdsbesteding geweest. Zelfs met een thesis waarvoor het onderwerp een eigen voorstel was, is het een gok of de doelstellingen wel verwezelijkt zullen worden. Ook is bij aanvang helemaal niet zeker of het onderwerp wel even interessant zal blijken te zijn als toen het werd voorgesteld. De voornaamste les die uit deze 10 maanden getrokken kan worden is dat men altijd voor verrassingen komt te staan, soms in positieve zin, soms in negatieve zin. Ook blijkt dat het aanpassen van programmatuur die door iemand anders ontwikkeld werd zeker geen makkelijk taak is. De frustrerende nachtelijke uren die het gevolg waren van moeilijk vindbare fouten zijn hier getuigen van.

Nu het werk van deze thesis beëindigd is, met een open einde dat hopelijk nog ingevuld zal worden in de toekomst, is het misschien passend de hoop uit te spreken dat de technologie waarop MUDs gebaseerd zijn meer en meer in de belangstelling zal komen. Hopelijk kan deze thesis een bijdrage leveren aan een dergelijke ontwikkeling. Nog al te dikwijls wordt een zeer interessante materie verworpen ten gevolge van de beperkte toepassingen die ervoor bestaan. MUDs zijn computerspelletjes die dikwijls met een negatief oog bekeken worden. Deze thesis kan misschien als secundair doel aantonen dat er meer mogelijk is. Enkel de toekomst kan hierover uitsluitsel geven.

Bijlage A

LPC syntax

```
<program>        ::= class <string_const> { <toplevel_decl> }
                   | entity <string_const> { <toplevel_decl> }
<toplevel_decls>
                 ::= <toplevel_decls> <toplevel_decl>
<toplevel_decl>  ::= inherit <opt_label> <inherit_string> ;
                   | <classl> <types> <list_decl> ;
                   | <classl> <types> <stars> <ident> ( <formals> )
                     <compound>
                   | <classl> <identifier> ( <formals> ) <compound>
<opt_label>      ::= NOTHING
                   | <identifier>
<inherit_string>
                 ::= <string_const>
                   | <inherit_string> + <string_const>
<classl>         ::= NOTHING
                   | <classl> <class_spec>
<class_spec>     ::= private
                   | static
                   | nomask
                   | varargs
<types>          ::= int
                   | float
                   | string
                   | object
                   | mapping
                   | mixed
                   | void
<stars>          ::= NOTHING
                   | <stars> *
<list_decl>      ::= <decl>
```

```
                 | <list_decl> , <decl>
<decl>          ::= <data_decl>
                 | <func_decl>
<data_decl>     ::= <stars> <identifier>
<func_decl>     ::= <stars> <identifier> ( <formals> )
<formals>       ::= NOTHING
                 | void
                 | <formal_list>
                 | <formal_list> ...
<formal_list>   ::= <formal>
                 | <formalsl> , <formal>
<formal>        ::= <types> <data_decl>
                 | <identifier>
<compound>      ::= { <decl_or_stmtl> }
<decl_or_stmtl> ::= NOTHING
                 | <decl_or_stmtl> <decl_or_stmt>
<decl_or_stmt>  ::= <class_specl> <types> <list_decl> ;
                 | <statement>
<statement>     ::= <list_exp> ;
                 | <compound>
                 | if ( <list_exp> ) <statement>
                 | if ( <list_exp> ) <statement> else <statement>
                 | do <statement> while ( <list_exp> ) ;
                 | while ( <list_exp> ) <statement>
                 | for ( <opt_list_exp> ; <opt_list_exp> ;
                   <opt_list_exp> ) <statement>
                 | switch ( <list_exp> ) <compound>
                 | case <exp> : <statement>
                 | case <exp> .. <exp> : <statement>
                 | default : <statement>
                 | break ;
                 | continue ;
                 | return <opt_list_exp> ;
                 | ;
<opt_list_exp>   ::= NOTHING
                 | <list_exp>
<list_exp>      ::= <exp>
                 | <list_exp> , <exp>
<exp>           ::= <cond_exp>
                 | <cond_exp> = <exp>
                 | <cond_exp> += <exp>
                 | <cond_exp> -= <exp>
                 | <cond_exp> *= <exp>
```

```
                   | <cond_exp> /= <exp>
                   | <cond_exp> %= <exp>
                   | <cond_exp> <<= <exp>
                   | <cond_exp> >>= <exp>
                   | <cond_exp> &= <exp>
                   | <cond_exp> ^= <exp>
                   | <cond_exp> |= <exp>
<cond_exp>       ::= <or_op_exp>
                   | <or_op_exp> ? <list_exp> : <list_exp>
<or_op_exp>      ::= <and_op_exp>
                   | <or_op_exp> || <and_op_exp>
<and_op_exp>     ::= <bitor_op_exp>
                   | <and_op_exp> && <bitor_op_exp>
<bitor_op_exp>   ::= <bitxor_op_exp>
                   | <bitor_op_exp> | <bitxor_op_exp>
<bitxor_op_exp>  ::= <bitand_op_exp>
                   | <bitxor_op_exp> ^ <bitand_op_exp>
<bitand_op_exp>  ::= <equ_op_exp>
                   | <bitand_op_exp> & <equ_op_exp>

<equ_op_exp>     ::= <rel_op_exp>
                   | <equ_op_exp> == <rel_op_exp>
                   | <equ_op_exp> != <rel_op_exp>
<rel_op_exp>     ::= <shift_op_exp>
                   | <rel_op_exp> < <shift_op_exp>
                   | <rel_op_exp> > <shift_op_exp>
                   | <rel_op_exp> <= <shift_op_exp>
                   | <rel_op_exp> >= <shift_op_exp>
<shift_op_exp>   ::= <add_op_exp>
                   | <shift_op_exp> << <add_op_exp>
                   | <shift_op_exp> >> <add_op_exp>
<add_op_exp>     ::= <mult_op_exp>
                   | <add_op_exp> + <mult_op_exp>
                   | <add_op_exp> - <mutl_op_exp>
<mult_op_exp>    ::= <cast_exp>
                   | <mult_op_exp> * <cast_exp>
                   | <mult_op_exp> / <cast_exp>
                   | <mult_op_exp> % <cast_exp>
<cast_op>        ::= <prefix_exp>
                   | ( <types> <stars> ) <cast_exp>
<prefix_exp>     ::= <postfix_exp>
                   | ++ <cast_exp>
                   | -- <cast_exp>
```

```
                | - <cast_exp>
                | + <cast_exp>
                | ! <cast_exp>
                | ~ <cast_exp>
<postfix_exp>   ::= <p2_exp>
                | <postfix_exp> ++
                | <postfix_exp> --
<p2_exp>        ::= <p1_exp>
                | <p2_exp> [ <list_exp> ]
                | <p2_exp> [ <opt_list_exp> .. <opt_list_exp> ]
<p1_exp>        ::= <int_const>
                | <float_const>
                | <string_const>
                | ( { <opt_arg_list_comma> } )
                | ( [ <opt_assoc_list_comma> ] )
                | <identifier>
                | ( list_exp )
                | <funcname> ( <opt_arg_list> )
                | catch ( <list_exp> )
                | lock ( list_exp )
                | <p2_exp> -> <identifier> ( <opt_arg_list> )
<arg_list>      ::= <exp>
                | <arg_list> , <exp>
<opt_arg_list>  ::= NOTHING
                | <arg_list>
                | <arg_list> ...
<opt_arg_list_comma>
                ::= NOTHING
                | <arg_list>
                | <arg_list> ,
<assoc_exp>     ::= <exp> : <exp>
<assoc_list>    ::= <assoc_exp>
                | <assoc_list> , <assoc_exp>
<opt_assoc_list_comma>
                ::= NOTHING
                | <assoc_list>
                | <assoc_list> ,
```

Bijlage B

Een aantal basis klassen en entiteiten

In deze bijlage wordt de broncode gegeven van een aantal klassen en entiteiten die belangrijk zijn in de programmeeromgeving die in deze thesis werd uitgewerkt. De syntaxbeschrijving voor LPC is in bijlage A gegeven. Om niet nodeloos in te moeten gaan op redelijk onbelangrijke details is voor sommige functies enkel een definitie en beschrijving gegeven in plaats van de volledige broncode. Ook worden "private" functies enkel vermeld indien nodig.

B.1 De driver entiteit: /kernel/init.c

De driver entiteit verzorgt de interactie tussen de DGD server en de LPC omgeving. De meeste functies in deze entiteit zullen dan ook enkel rechtstreeks vanuit de server opgeroepen worden. Om veiligheidsredenen zijn veel van deze functies als "static" gedefiniëerd. De server is namelijk in staat om een dergelijke functie toch op te roepen. Dit kan verklaard worden op basis van het feit dat de driver entiteit als interface eigenlijk gedeeltelijk verweven is met de server.

```
/*
 * FILE:        init.c
 * DESCRIPTION: System startup code and driver interface
 */

#include <kernel.h>             /* kernel specific definitions/macros */
#include <status.h>             /* definitions for the status() kfun */

entity "init" {
    private mapping     map;    /* initial name conversion table */

    /*
     * NAME:        _F_create()
     * DESCRIPTION: creator function
```

```
 */
void _F_create() {
    /* load the initial (static) name conversion mapping */
    map = (["object": "/core/object",
            "driver": "/kernel/init",
            "named": "/kernel/named",
            "savable": "/core/savable",
            "trie": "/core/trie",
          ]);
}

/*
 * NAME:        writek()
 * DESCRIPTION: write a message to the system log
 */
void writek(string s)

/*
 * NAME:        initialize_named()
 * DESCRIPTION: initialize the name conversion daemon
 */
private void initialize_named() {
    writek("Starting up name conversion daemon...\n");

    if (!NAMED->tablesize()) {      /* only if needed */
        int               i;
        string        *names, *files;

        writek("Loading initial name conversion table...\n");
        names = map_indices(map);   /* list of indices */
        files = map_values(map);    /* list of values */
        for (i = sizeof(names); i; )/* process all indices */
            /* for each name, the (name, filename) pair will be *
             * added to the dynamic conversion table            */
            NAMED->put_entry(names[--i], files[i], 0);
        NAMED->save();              /* force object dump to file */
    }
}

/*
 * NAME:        initialize()
 * DESCRIPTION: system boot sequence
 */
```

```
static void initialize() {
    initialize_named();
    writek("INIT: System up and running.\n");
}

/*
 * NAME:        restored()
 * DESCRIPTION: system boot sequence (from state dump)
static void restored()
 * NAME:        path_ed_read()
 * DESCRIPTION: translate a path for an editor read command
string path_ed_read(string path)
 * NAME:        path_ed_write()
 * DESCRIPTION: translate a path for an editor write command
string path_ed_write(string path)
 */

/*
 * NAME:        path_object()
 * DESCRIPTION: translate an object name to a file name
 */
string path_object(string name) {
    string       file;

    file = NAMED->get_entry(name);  /* try named first */
    return file ? file : map[name]; /* if needed, use static map */
}

/*
 * NAME:        path_include()
 * DESCRIPTION: translate an include path
string path_include(string file, string path)
 * NAME:        compile_object()
 * DESCRIPTION: used to create virtual objects
static object compile_object(string name)
 * NAME:        recompile()
 * DESCRIPTION: used to recompile objects
static void recompile(object ob)
 */

/*
 * NAME:        telnet_connect()
 * DESCRIPTION: return a user object
```

```
     */
    static object telnet_connect() {
        object       ob;

        /* make a new telnet object and return it */
        ob = clone_object("telnet connection");
        return ob;
    }

    /*
     * NAME:        binary_connect()
     * DESCRIPTION: return a user object
     */
    static object binary_connect() {
        object       ob;

        /* make a new HTTP processor object and return it */
        ob = clone_object("HTTP connection");
        return ob;
    }

    /*
     * NAME:        log_error()
     * DESCRIPTION: log a runtime error
    static void log_error(string err, int caught)
     * NAME:            compile_log()
     * DESCRIPTION: return path of a secondary compile log
    string compile_log(string file)
     */
}
```

B.2 De naam conversie: /kernel/named.c

Deze entiteit vormt de basis voor de naam conversie die in de thesis implementatie van DGD nodig is. De reden om hier een entiteit van te maken volgt voornamelijk uit de observatie dat het zinloos is meerdere exemplaren van die object te hebben. Uiteraard zijn alle andere objecten in de “trie” structuur instantiaties van de “trie” klasse omdat hier meedere exemplaren van moeten kunnen bestaan. Het enige doel van deze entiteit is dan ook basis spelen voor de gehele trie–structuur, en aanvragen voor naam conversie verwerken. Het is de wortel van de conversie structuur.

```
/*
```

```
 * FILE: named.c
 * DESCRIPTION: Name conversion class
 */

#include <kernel.h>             /* kernel specific definitions/macros */

entity "named" {
    inherit "trie";             /* the root is a trie-element itself */

    /*
     * NAME:        create()
     * DESCRIPTION: creator function
     */
    static void create() {
        ::setup("/kernel/tables/named-", 0);    /* load the table */
    }
}
```

B.3 De trie klasse: /core/trie.c

Deze klasse implementeerd een redelijk complexe structuur. Een klassieke trie kan men zich best voorstellen als een boomstructuur waarbij elke vertakking met een letter geassociëerd wordt. Op deze manier kunnen strings opgeslagen worden door gewoon het unieke pad te vormen langs de letters in de string. Ook geeft dit de mogelijkheid heel makkelijk te testen of een bepaalde string in de trie aanwezig is. Zodra een knoop gevonden wordt die slechts één vertakking heeft, weet men of de string aanwezig is of niet.

Voor de naam conversie wordt er een aangepaste versie van een trie structuur toegepast. De basis van de structuur is een hashtabel. De sleutels zijn hier de arbitraire namen, en de waarden de bestandsnamen die erbij horen. Het aspect van tries komt slechts tot uiting wanneer de hashtabel vol begint te geraken. Dan wordt namelijk een deel van de hashtabel verschoven naar een volgend niveau. Deze verschuiving gebeurt op basis van het principe van tries, namelijk door een opsplitsing te maken op basis van een letter in de naam.

```
/*
 * FILE: trie.c
 * DESCRIPTION: Trie structure class
 */

#include <kernel.h>
#include <limits.h>
#include <trie.h>
#include <type.h>
```

```
class "trie" {
    inherit "savable";

    private int         depth;  /* level in the trie */
    private int         tabsz;  /* number of elements in table */

    mapping     count;          /* class counters */
    mapping     level;          /* level pointers and counters */
    mapping     table;          /* data on this level */

    /*
     * NAME:        setup()
     * DESCRIPTION: setup function
     */
    void setup(string m, int d) {
        set_savename(m);

        if (!load()) {          /* if it can't be loaded, initialize */
            count = ([]);
            level = ([]);
            table = ([]);
        }

        depth = d;              /* used to know wich letter to check */
        tabsz = map_sizeof(table);
    }

    /*
     * NAME:        save()
     * DESCRIPTION: override save() in SAVABLE
     */
    void save() { ::save(); }

    /*
     * NAME:        tablesize()
     * DESCRIPTION: return the size of the conversion table
     */
    int tablesize() { return tabsz; }

    /*
     * NAME:        put_entry()
     * DESCRIPTION: Put an entry in the trie structure
```

```
 */
mixed put_entry(string s, mixed data, int force_save) {
    mixed       old;
    object       ob;

    if (old = table[s]) {   /* found in table already so replace */
        table[s] = data;
        if (force_save) ::save();
        return old;         /* return the old value */
    }
    /* we check the letter on position ``depth'' if possible, and *
     * if a lower level exists for it, pass on the put action     */
    if (strlen(s) > depth &&
        typeof(old = level[s[depth]]) == T_ARRAY) {
        if (!(ob = old[TRIE_OBJ]))  /* the lower level isn't loaded */
            ob = old[0] = "trie"->clone(old[TRIE_TAB], old[TRIE_LVL]);
        return ob->put_entry(s, data, force_save);
    }
    if (tabsz >= LIMITSZ) {
        int                i, limit;
        mixed        *b;
        string        *a;

        /* this loop will reduce the size of the mapping by moving *
         * entries with a common letter in the ``depth'' position  *
         * to a new level below this one - count already contains  *
         * the number of names per letter as we need it             */

        a = map_indices(table);
        b = map_values(table);
        i = tabsz - 1;
        limit = LIMITSZ / map_sizeof(count);

        while (i > 0) {
            int        j, c;
            string        t;

            if (strlen(t = a[i]) <= depth) continue;
            if ((j = count[c = t[depth]]) >= limit) {
                string        fn;

                fn = " ";
                fn[0] = c;
```

```
                    fn = savename() + fn;
                    tabsz -= j;
                    count[c] = 0;
                    level[c] = ({ ob = "trie"->clone(fn, depth + 1), fn,
                                  depth + 1 });
                    while (j) {
                        if (strlen(t = a[--i]) <= depth) continue;
                        else --j;
                        ob->put_entry(t, b[i], 0);
                        table[t] = 0;
                    }
                    ob->save();
                } else
                    while (j--) --i;
            }
        }
        if (strlen(s) > depth) count[s[depth]]++;   /* update count */
        table[s] = data;                            /* add entry */
        tabsz++;
        if (force_save) ::save();
        return 0;
    }

    /*
     * NAME:        get_entry()
     * DESCRIPTION: Get an entry from the trie structure
     */
    mixed get_entry(string s) {
        mixed       data;
        object      ob;

        if (data = table[s]) return data;   /* found it */
        if (strlen(s) <= depth ||           /* next next level */
            typeof(data = level[s[depth]]) != T_ARRAY)
            return 0;
        if (ob = data[TRIE_OBJ]) return ob->get_entry(s);
        ob = "trie"->clone(data[TRIE_TAB], data[TRIE_LVL]);
        return ob->get_entry(s);
    }
}
```

Bibliografie

[A+89] Richard Acott et al. *AberMUD: A guide to the system and its use*, 1989.

[Bjö89] Anders Björnerstedt. Secondary storage garbage collection for decentralized object–based systems. In D. Tsichritzis, editor, *Object Oriented Development*. Centre Universitaire d'Informatique, University of Geneva, 1989.

[Boo94] Grady Booch. *Object-Oriented Analysis And Design With Applications*. Benjamin Cummings, 1994.

[Bru94] Amy Bruckman. Programming for fun. muds as a context for collaborative learning. Technical report, MIT Media Lab, 1994.

[CN93] Pavel Curtis and David A. Nichols. Muds grow up: Social virtual reality in the real world. Technical report, Xerox PARC, 1993.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison Wesley, 1992.

[DM93] Bart De Moen. *Theorie der programmeertalen, deel Algol 60*. Vlaams Technische Kring vzw, 1993. cursus 403.

[HZ93] Robert Henderson and Benjamin Zorn. A comparison of object–oriented programming in four modern languages. Technical Report CU-CS-641-93, University of Colorado in Boulder, 1993.

[Kay92] Erik Kay. An overview of the history of mudos. bestand doc/History.MudOS uit ftp://ftp.lysator.liu.se/pub/lpmud/drivers/MudOS_0.9.19.tar.gz, 1992.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[MO92] James Martin and James J. Odell. *Object-Oriented Analysis and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[P+93] Lars Pensjö et al. Lpmud, a programmable multi user game (pre-release). ftp://sunsite.unc.edu/pub/Linux/games/muds/lpmuddoc.ps.z, 1993.

[R+91] James Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[SM88] Sally Shlaer and Stephen J Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. 1988.

[SW90] Jennifer Stone and Rusty C. Wright. *A brief guide to TinyMUD*, 1990. bestand uit ftp://gatekeeper.dec.com/.b/games/tinymud_doc.tar.Z.

[US91] David Ungar and Randall B. Smith. Self: The power of simplicity. Op World Wide Web te vinden als URL http://self.stanford.edu/papers/self-power.html, 1991.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. Bedoeld om gepubliceerd te worden in de proceedings van de 1992 International Workshop on Memory Management (St. Malo, France, September 1992), 1992.