Hoofdstuk 9

Objectgeorienteerde programmeertalen

Een computerprogramma is een serie opdrachten die door de computer wordt uitgevoerd. Dat is een juiste omschrijving, maar er valt meer over te zeggen dan dit.
Door het gebruik van hogere programmeertalen kunnen deze opdrachten in voor mensen begrijpelijke taal worden beschreven. In deze programmeertalen kun je bovendien gebruik maken van handige structuren zoals de keuzestructuur en de herhalingsstructuur. Verder is het mogelijk om grote programma's op te splitsen in deeltaken. Die deeltaken noemen we procedures (Pascal), subroutines (Basic) of methodes (Java). Objectgeorienteerde talen gaan nog veel verder in het opdelen en structureren van programma's. Er wordt gebruik gemaakt van objecten. Dat zijn kant en klare stukken code die de programmeur kan gebruiken als de bouwstenen voor zijn programma. Zo'n object heeft een duidelijk omschreven functie en een duidelijke gebruiksaanwijzing voor de programmeur. Een object kan duizenden regels code bevatten met vele procedures. De kracht ligt in het feit dat de programmeur niet hoeft te weten hoe deze code precies werkt, die kan namelijk door iemand ander gemaakt zijn, maar hij kan erop vertrouwen dat het doet wat het belooft. Dat maakt het programmeren makkelijker, of beter gezegd, je kunt met dezelfde moeite complexere programma's maken. Uiteindelijk moet je concluderen dat objectgeorienteerd programmeren noodzakelijk werd om de zeer complexe programma's van tegenwoordig nog te kunnen maken.

9.1 Communicatie met programma's

Via het toetsenbord en tekstscherm ...

Er zijn zeer veel verschillende manieren waarop computerprogramma's invoer mogelijk maken voor de gebruiker van het programma. Bij oudere programma's, laten we zeggen programma's van vóór het Windows tijdperk werd de invoer voor een programma gegeven door het intypen van tekst. De communicatie liep ongeveer zo:

- Via een tekstscherm gaf de gebruiker een opdracht om het programma te starten.
- Het programma vroeg om invoer van de gebruiker.
- De gebruiker gaf de invoer door iets in te typen, gevolgd door de "enter-knop" (enter betekent: voer in).
- Het programma verwerkte de invoergegevens, en gaf een uitvoer vaak weer op hetzelfde tekstscherm.
- Het programma werd beëindigd, of vroeg weer om een nieuwe invoer, enzovoort.

Bij zo'n manier van communiceren ligt het initiatief grotendeels bij het programma. Het programma bepaalt wanneer er invoer nodig is, en meestal ook in wat voor vorm die gegeven moet worden. Bovendien is het zo dat de gebruiker vaak een groot aantal opdrachtcomando's die als invoer gegeven kunnen worden, uit zijn hoofd moet kennen om vlot met programma te kunnen werken. Gebruiksvriendelijk kun je deze manier van communiceren niet noemen, vooral als je het vergelijkt met de manier waarop we tegenwoordig met programma's werken.

... naar een grafische gebruikers interface

Voor 1980 waren computers niet krachtig genoeg om een compleet beeldscherm pixel voor pixel te tekenen. De eerste PC had een tekstscherm met 25 regels van 80 letters. Je ziet dit nog steeds als je onder Windows een MS-Dos-venster opent, zoals je gedaan hebt in

hoofdstuk 5. Grafische programma's waren zeldzaam: de Apple II kon met gekleurde blokjes werken!
In 1980 introduceerde het bedrijf Xerox een beeldscherm waarvan de afzonderlijke pixels door de computer konden worden getekend. Op het nieuwe scherm kon je invoervelden tekenen. Naast het normale toetsenbord kon je ook een merkwaardig apparaatje gebruiken, dat later "muis" genoemd werd. Dit idee werd in 1984 door het bedrijf Apple als eerste commercieel toegepast in de Macintosh. De Mac sloeg direct aan bij het publiek en werd een groot succes. De *grafische gebruikersinterface* (Engels: *GUI*, ofwel Grafical User Interface) was een feit.
De concurrerende computer PC van IBM volgde aanvankelijk aarzelend deze trend, en werd vanaf eind jaren tachtig ook uitgerust met een besturingssyteem met een grafische interface: Windows van Microsoft.

De voordelen van een grafische gebruikersinterface zijn groot. De gebruiker kan op veel verschillende manieren opdrachten geven aan het systeem. Invoer kan plaatsvinden door tekst in te typen in een tekstveld, waarvan de functie duidelijk staat aangegeven, maar ook door te klikken op een knop, het kiezen van een optie uit een menu, of door het verslepen van icoon of ander object. De gebruiker hoeft niet meer allerlei commando's uit het hoofd te weten. De interface maakt op een intuitieve manier duidelijk wat de verschillende mogelijkheden zijn voor de gebruiker. De communicatie wordt door deze aanpak niet alleen duidelijker, je kunt programma's ook zo maken dat het initiatief in de communicatie bij de gebruiker komt te liggen.

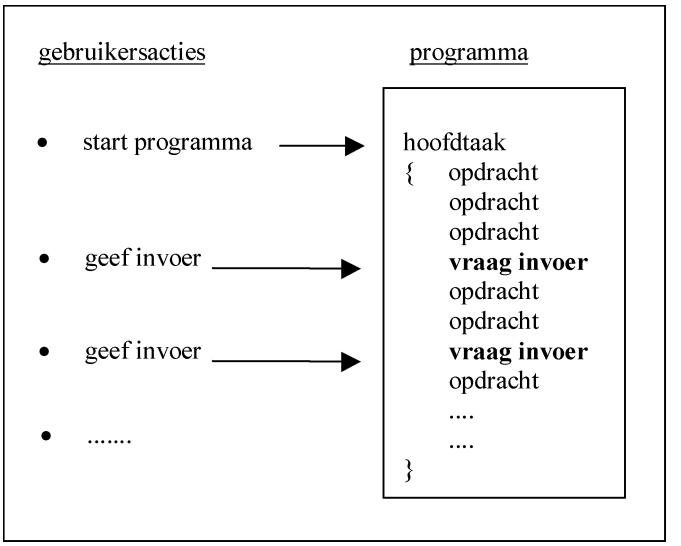
De ontwikkeling van programma's met een grafische interface hangt samen met de ontwikkeling van multitasking mogelijkheden bij besturingssystemen (meerdere processen kunnen "tegelijkertijd" draaien). Bij een GUI is dit nodig. Als het programma bezig is met het uitvoeren van een taak, moet het tegelijkertijd ook kunnen reageren op een muisklik. Dit is besproken in hoofdstuk 4.

Het zal duidelijk zijn dat de GUI een belangrijke factor is geweest in de ontwikkeling van de computer tot een medium dat door bijna iedereen wordt gebruikt. De grafische interface is nu een vanzelfsprekendheid voor iedere computergebruiker. Daar staat tegenover dat de software die zo'n grafische interface mogelijk maken, veel complexer is dan de oude tekstgebaseerde programma's, waardoor programmeurs nieuwe gereedschappen moesten ontwikkelen. Zo zijn de nieuwe OO-talen ontstaan.

Gebeurtenisgestuurde programma's

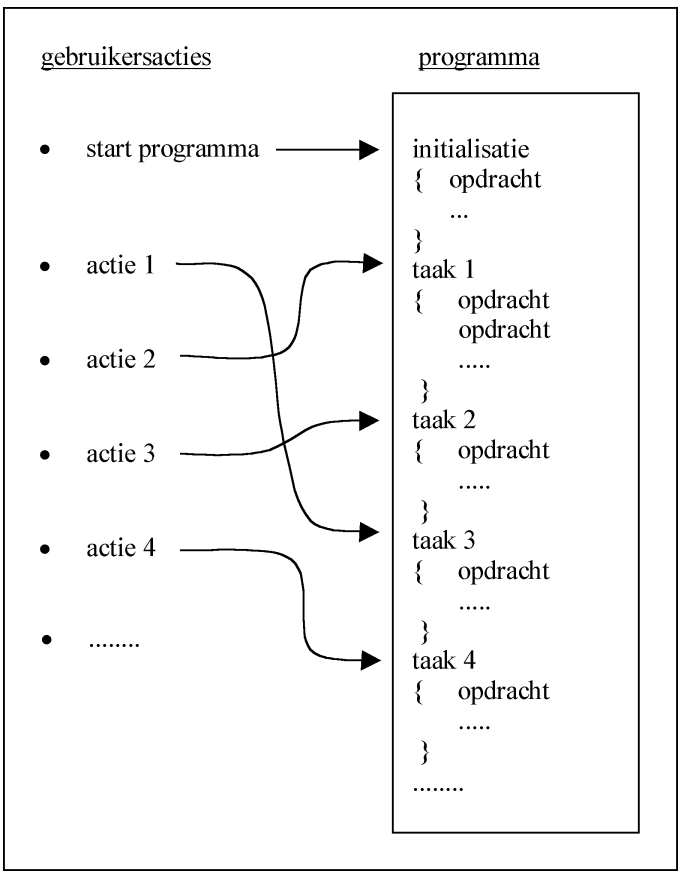
Het basispatroon van een ouder programma zonder GUI, kan het best worden omschreven als een serie opdrachten die achtereenvolgens worden uitgevoerd, die aan de gebruiker op bepaalde vastgelegde momenten om invoer of sturing vragen. Dit soort programma's kun je op een eenvoudige manier in beeld brengen met behulp van een structuurdiagram (PSD), zoals we in hoofdstuk 7 gezien hebben. In figuur 1 zie je de communicatie met de gebruiker.

Het basispatroon van een modern programma met een GUI is anders. Het enige wat het programma doet als het wordt gestart, is een beginscherm laten zien. Vervolgens wacht het op een *gebeurtenis*, een handeling van de gebruiker. Zo'n gebeurtenis kan zijn: het drukken op een knop, of het maken van een keuze uit een menu, enzovoort. In het Engels heet een gebeurtenis een *event*.



Figuur 1 Acties van een gebruiker bij een programma zonder een GUI

Bij elke mogelijke gebeurtenis zal het programma reageren met het uitvoeren van een reeks opdrachten. Daarna wordt het scherm aangepast en wacht het programma op een nieuwe gebeurtenis. Daarom noemt men dit soort programma's *gebeurtenisgestuurd*, in het Engels '*event driven*'. In figuur 2 zie je het communicatiepatroon.



Figuur 2 Acties van een gebruiker bij een programma met een GUI

Is met de komst van deze nieuwe generatie programma's het basispatroon verouderd, en niet meer toepasbaar? Het antwoord is nee. Binnen een gebeurtenisgestuurd programma zijn de afzonderlijke taken die worden uitgevoerd bij een bepaalde gebeurtenis, toch weer te beschrijven als een serie opdrachten. Dus binnen zo'n taak kun je de PSD's weer nuttig gebruiken. Je bent in hoofdstuk 7 en 8 dus niet bezig geweest met een verouderde manier van programmeren.

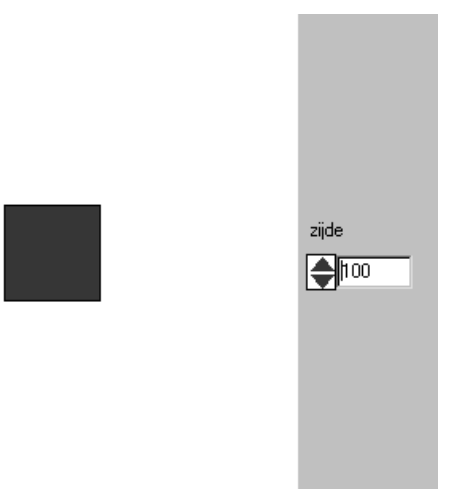
Omdat gebeurtenisgestuurde programma's een ander communicatiepatroon hebben met de gebruikers, zijn ze ook op een andere manier geprogrammeerd. In oude programmeertalen als Basic, Pascal en C zijn dit soort programma's niet makkelijk te maken. Een nieuwe generatie van objectgeoriënteerde talen is hier geschikt voor: C++, Java, Delphi en Visual Basic.
Een belangrijk begrip in dit soort talen is *event handling*, het Engelse woord voor *gebeurtenisafhandeling*. De programmeur moet bepalen welke opdrachten er moeten worden uitgevoerd bij een bepaalde gebeurtenis.

In de digitale practica die bij dit hoofdstuk gebruikt worden, kun je zelf gebeurtenis-gestuurde en objectgeorienteerde pragramma's maken. Er zijn digitale practica voor JavaLogo, maar ook voor de taal Delphi, de objectgeorienteerde opvolger van Pascal. In de rest van dit hoofdstuk gaan we in op de grote lijnen en achtergronden van deze manier van programmeren. Daarbij gebruiken we enkele concrete voorbeelden uit deze digitale practica.

9.2 JavaLogo programma's met eventhandling

De eerste serie digitale practica bij JavaLogo ging over programma's voor het maken van tekeningen. Als zo'n programma wordt uitgevoerd, dan wordt de tekening op het scherm gezet en daarmee is het programma klaar. De gebruiker kan niet communiceren met het programma, maar het alleen uitvoeren. In de tweede serie digitale practica leer je programma's maken die dat wel kunnen. Met behulp van knoppen en invulvakjes kun je de tekening veranderen of laten bewegen. Er is ook interactie mogelijk met behulp van de muis, waardoor tekeningen bijvoorbeeld kunnen worden versleept.

We bekijken het applet van figuur 3. Aan de rechterkant zie je een zogenaamde invoervariabele staan. In programma 1 zie je de code van deze applet.



Figuur 3 Applet met invoervariabele

```
import logotekenap.*;

public class Vierkant extends TekenApplet
{
1  InvoerVariabele zijdeInv;
   double zijde;

   public void initialiseer()
2  {  zijdeInv = new InvoerVariabele("zijde",0,400,100);
3     maakZichtbaar(zijdeInv);
      zijde = 100;
   }

   public void tekenprogramma()
   {  stap(-zijde/2,-zijde/2);
      vierkant(zijde);
   }

   void vierkant(double z)
   {  vulAan("rood");
      for(int i=0 ; i<4 ; i++)
      {  vooruit(z);
         rechts(90);
      }
      vulUit();
   }

4  public void invoerVarActie(InvoerVariabele iv)
5  {  zijde = zijdeInv.geefWaarde();
6     tekenOpnieuw();
   }
}
```

Programma 1

In de code van het programma zie je nieuwe dingen. We gaan ze één voor één af.

1 Het programma begint met de regel:

```
InvoerVariabele zijdeInv;
```

Dit is een *declaratie* van een variabele met de naam zijdeInv. Je zou hier iedere naam kunnen gebruiken die je maar wilt, maar wij hebben gekozen voor zijdeInv.
In hoofdstuk 6 heb je ook al met variabelen gewerkt. Dat waren eenvoudige getallen, zoals double, int, soms ook een String (een rij letters zoals "rood").
Variabelen kunnen dus eenvoudige getallen zijn (zoals double of int), maar ook een String (een rij letters zoals bijvoorbeeld "rood") en andere complexe 'dingen' met allerlei eigenschappen zijn. Zo'n soort variabele noemen we een *object*. Een InvoerVariabele is een voorbeeld van een object. Programmeren met objecten noemen *we objectgeoriënteerd programmeren*.

2 We zien in de methode initialiseer() de regel:

```
zijdeInv = new InvoerVariabele("zijde",0,400,100);
```

Voordat een object gebruikt kan worden in een programma moet het gemaakt worden. Dat gebeurt in deze regel. Met new InvoerVariabele(...) wordt er een nieuw object van het type InvoerVariabele gemaakt.
We noemen dit een *constructor*. Dit object heet nu zijdeInv (zijdeInv = ...). Met behulp van de String en de getallen die tussen de haakjes staan, worden bovendien een aantal eigenschappen van het object vastgelegd:

- De String "zijde" bepaalt de tekst die op het label komt te staan.
- Het eerste getal is de kleinst mogelijke waarde van de InvoerVariabele.
- Het tweede getal is de grootst mogelijke waarde van de InvoerVariabele.
- Het derde getal is de beginwaarde van de InvoerVariabele.

3 Na de creatie volgt de opdracht:

```
maakZichtbaar(zijdeInv);
```

Met behulp van deze opdracht wordt de InvoerVariabele zichtbaar gemaakt op een grijs paneel aan de rechterkant van het applet.

4 Wanneer de gebruiker van het programma de InvoerVariabele een andere waarde geeft, dan moet het programma reageren op deze gebeurtenis met een actie. Wat het programma in dat geval moet doen staat beschreven in de *methode*:

```
public void invoerVarActie(InvoerVariabele iv)
{  zijde = zijdeInv.geefWaarde();
   tekenOpnieuw();
}
```

Zo'n methode noemen we een *event handler* omdat deze bepaalt wat er moet gebeuren bij een event, een gebeurtenis. Wanneer de gebruiker iets verandert aan een Invoervariabele, dan wordt deze methode aangeroepen en uitgevoerd.

5 In de regel:

```
zijde = zijdeInv.geefWaarde();
```

wordt de waarde van de InvoerVariabele zijdeInv opgevraagd en toegekend aan de variabele *zijde.*
De methode geefWaarde() hoort bij objecten van de klasse InvoerVariabele en moet dus altijd gebruikt worden in combinatie met zo'n object. Vandaar de uitdrukking zijdeInv.geefWaarde(). Er valt over dit soort zaken nog veel te vertellen, maar we zullen hier op dit moment niet verder op in gaan. Voor nu is het voldoende om te weten hoe je de waarde van een InvoerVariabele kunt opvragen.

6 De laatste opdracht spreekt voor zich:

```
tekenOpnieuw();
```

De opdrachten in de methode tekenprogramma() worden opnieuw uitgevoerd. Het resultaat is dat de tekening van het vierkant opnieuw gemaakt wordt, maar nu met de nieuwe waarde van de variabele zijde.

Andere eventhandlers bij JavaLogo

Behalve events van invoervariabelen kan JavaLogo nog enkele andere events afhandelen. Zo is het mogelijk om animaties te starten met een animatieknop. Deze eventhandler heet “animatie” en ziet er bijvoorbeeld zo uit:

```
public void animatie()
{  while ( animatieLopend() )
   {  hoek = hoek + 1;
      tekenOpnieuw();
   }
}
```

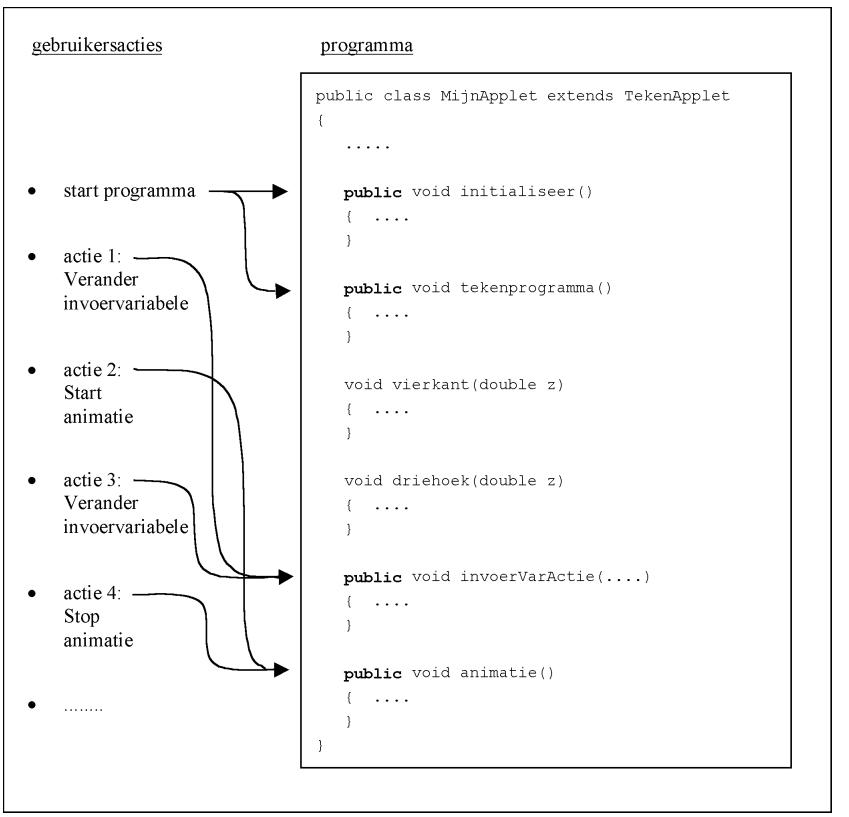
Wat hier gebeurt is, kort gezegd, dat een hoek steeds groter wordt gemaakt. Deze hoek wordt in een tekening gebruikt, waardoor de tekening steeds met een andere hoek wordt getekend. Het resultaat is een draaiend plaatje. Hoe dit precies werkt kun je aan weet komen in het digitaal practicum.

Ook is het mogelijk omde tekeningen aan te passen met behulp van muis-events, bijvoorbeeld klikken en slepen. Een muis-eventhandler in JavaLogo ziet er zo uit:

```
public void muisSleepActie()
{  beginX = beginX + geefSleepdx();
   beginY = beginY + geefSleepdy();
   tekenOpnieuw();
}
```

Nogmaals: communicatie tussen gebruiker en programma

We vestigen we nog een keer de aandacht op het communicatiepatroon van een willekeurig JavaLogo-programma met de gebruiker. In paragraaf 1 hebben dat onderwerp al even kort aangeroerd bij de bespreken van gebeurtenisgestuurde programma's.



Figuur 4 Interactie bij een JavaLogo-programma

In figuur 4 zie je duidelijk hoe de (invoer)acties van de gebruiker, gekoppeld zijn aan de publieke methodenvan het programma. De methoden initialiseer() en tekenprogramma() worden uitgevoerd bij het starten van het programma. Daarna bepalen de gebruikersacties welke event handlers wordt aangeroepen. De gebruiker heeft het initiatief in de communicatie.

9.3 Objectgeoriënteerd programmeren in Java

Java is een objectgeoriënteerde programmeertaal. Toch ben je in de eerste digitale practica van JavaLogo nauwelijks in aanraking gekomen met écht objectgeoriënteerd programmeren. Het enige objectgeoriënteerde aan je programma was de beginregel:

```
public class MijnTekenprog extends TekenApplet
{   ……
    ….
}
```

Een objectgeoriënteerde taal werkt met objecten. Er zijn objecten van verschillende soorten. Die soorten noemen we klassen (Engels: class). In je programma was tot nu toe steeds maar één object van één enkele klasse, in het voorbeeld hierboven heet deze klasse MijnTekenprog. In grotere programma's wordt meestal gewerkt met meer dan één klasse en veel objecten. Omdat jouw programma's er maar één hadden, merkte je nog weinig van objectgeoriënteerd programmeren.

Bij het gebruik van InvoerVariabelen in de tweede serie met digitale practica verandert dat. Invoervariabelen zijn ook objecten en in programma 2 is er sprake van twee objecten. Je eigen programma is een object en er is een InvoerVariabele: zijdeInv.
Programma's met meer dan één object verschillen van programma's met een enkel object op een paar punten. Ten eerste zijn de methode-aanroepen soms anders. Van ieder object kun je methoden aanroepen. De methoden van je eigen klasse kun je gewoon met hun naam aanroepen: vierkant(100), driehoek(50), enzovoorts.
Wanneer je de methoden van andere objecten wilt aanroepen, moet je de naam van het object voor de methode-aanroep zetten, met een punt ertussen: je schrijft daarom breedteInv.geefWaarde() om de waarde op te vragen van de InvoerVariabele breedteInv en hoogteInv.geefWaarde() om de waarde op te vragen van de InvoerVariabele hoogteInv.
De eigen klasse - je tekenprogramma - schrijf je zelf, maar bij InvoerVariabele hoeft dat niet. Dat is al gedaan door een andere programmeur. Die heeft een programma geschreven dat er zo uit ziet:

```
public class InvoerVariabele extends Component
{   ……

    public InvoerVariabele(String naam, int min, int max, int beginwaarde)
    {   // code voor het maken van een InvoerVariabele
    }

    public int geefWaarde()
    {   // code voor het opvragen van de waarde van de InvoerVariabele
    }

    /* nog veel meer, bijvoorbeeld event handlers voor het klikken
     * op de pijlen
     */
}
```

Programma 2

Er is een ‘public class InvoerVariabele’, dat wil zeggen dat publiek gebruikt mag worden. Jij bent zelf iemand uit ‘het publiek’ en je mag de klasse gebruiken.
InvoerVariabele is een uitbreiding van Component, dat is de verzamelnaam voor alle onderdelen van een user-interface, zoals knoppen, menu’s, scrollbars, enzovoorts. Een InvoerVariabele is een ‘component’ in de user-interface van je programma.

Verder staat er een publieke methode:

```
InvoerVariabele(String naam, int min, int max, int beginwaarde)
```

Deze methode is in programma 2 aangeroepen om een InvoerVariabele te maken met de tekst “breedte”, een minimum van 0, een maximum van 400 en de beginwaarde 200:

```
breedteInv = new InvoerVariabele("breedte",0,400,200);
```

Aan het woordje *new* kun je al zien dat dit een bijzondere methode-aanroep is. Omdat je er objecten mee kunt maken, heet het een constructor. Een constructor roep je altijd aan met new ervoor.

De programmeur heeft ook een publieke methode geefWaarde geschreven:

```
public int geefWaarde()
```

Dat wil zeggen dat het publiek deze methode mag gebruiken om de waarde van de InvoerVariabele op te vragen. In programma 2 gebeurt dat in de methode invoerVarActie().

Om met objecten van de klasse InvoerVariabele te kunnen werken is het voldoende dat je weet welke publieke methoden je kunt gebruiken. Hoe de code van InvoerVariabele er uitziet doet er verder niet toe, als het maar werkt! Het kan best zijn dat de programmeur urenlang heeft zitten zwoegen om de werking van de pijltjes goed te krijgen. Voor jou is dat niet belangrijk, je gebruikt het kant en klare programma.
De InvoerVariabele wordt daardoor een black box met een gebruiksaanwijzing. Hoe het werkt hoef je niet te weten, als de publieke methoden in de gebruiksaanwijzing het maar doen.
De gebruiksaanwijzing wordt vastgelegd in een zogenaamde *API* (*Application Programming Interface*). De API van Java is in html gemaakt, je kunt een browser gebruiken om de API te bekijken. Van de klasse InvoerVariabele is een API beschikbaar, evenals van de klasse Tekenapplet die we gebruiken in onze programma's.

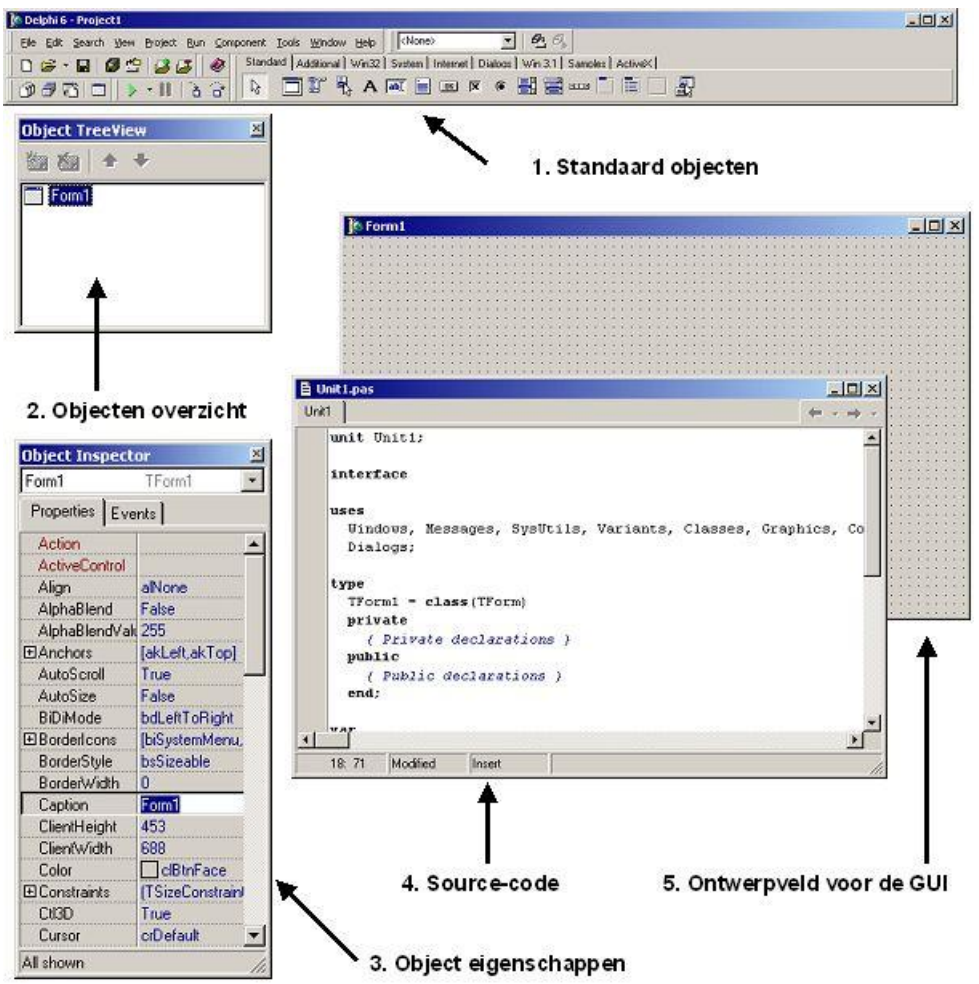
Het gebruiken van kant en klare klassen is een belangrijk voordeel van objectgeoriënteerd programmeren. Het wordt wel ‘herbruikbaarheid van code’ genoemd. Voor een programmeur scheelt dit een hoop werk, je hoeft niet steeds opnieuw het wiel uit te vinden!

9.4 Objectgeorienteerd programmeren in Delphi

Delphi (eigenlijk Object Pascal) is een objectgeorienteerde programmeertaal vergelijkbaar met Java. Net als in Java zijn er al heel veel objecten kant en klaar aanwezig. Dat geldt eigenlijk voor alle objectgeorienteerde programmeertalen zoals ook Visual Basis en C++. De meest in het oog springende objecten van al deze programmeertalen zijn de GUI-elementen, zoals bijvoorbeeld knoppen, invulvelden, labels, menubalken en dialoogvensters.
In de digitale practica waarin je praktisch kennis maakt met deze programmeertaal, maakt je gebruik van een zogenaamde visuele programmeeromgeving, die het gebruik van objecten bij het maken van programma’s nog makkelijker maakt.

Visuele programmeeromgevingen

Objectgeorienteerde programmeertalen gebruiken vaak een visuele programmeeromgeving. Bij de JavaLogo gebruikten we geen visuele programmeeromgeving. We maakten wel gebruik van kant en klare objecten, maar we schreven zelf de code om deze objecten te kunnen gebruiken. In onze code moesten we de gewenste objecten zelf declareren en via een constructor-aanroep maken. Bovendien was het nodig om het object op de interface te plaatsten. In de visuele programmeeromgeving van Delphi gaat dat veel makkelijker. In figuur 4 zie het openingsscherm van Delphi.



Figuur 5 Openingsscherm van de visuele programmeeromgeving van Delphi

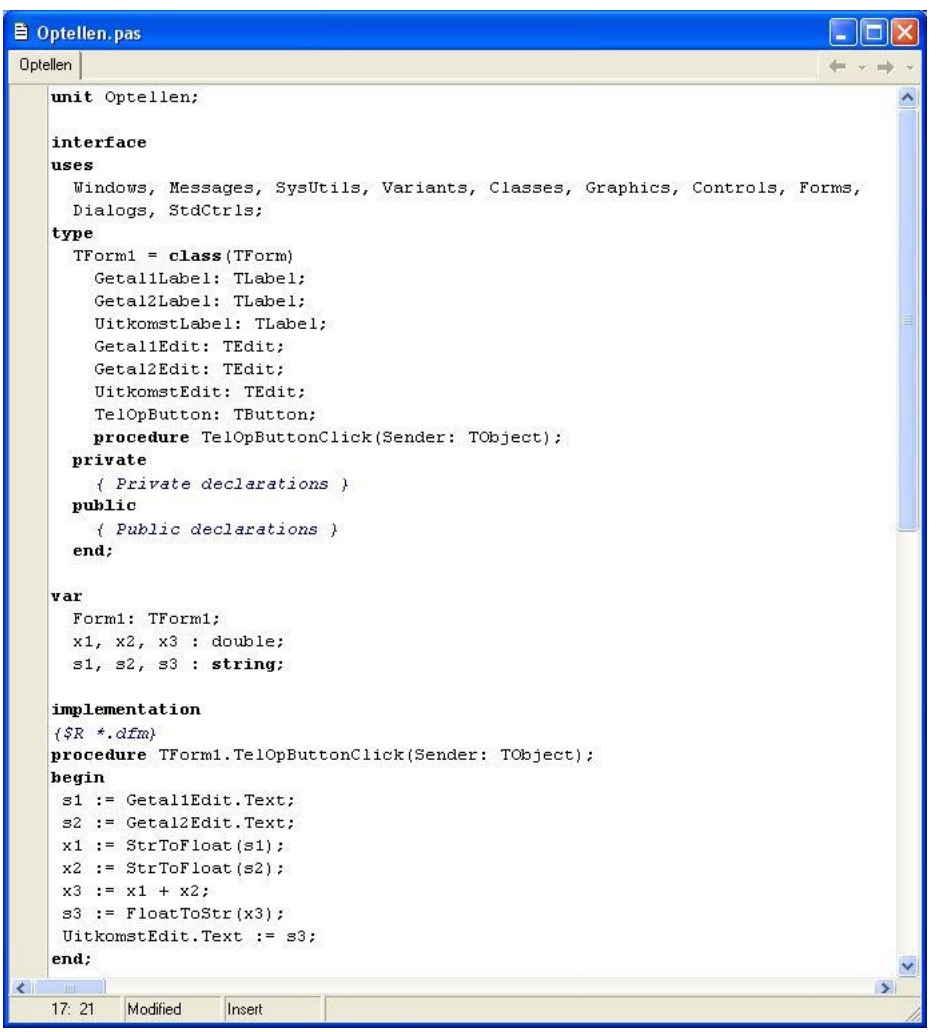
Op het ontwerpveld voor je nieuwe programma teken je de verschillende interfaceobjecten met behulp van een toolbar met standaardobjecten. Tijdens dit tekenen wordt de bijbehorende programma-code automatische gegenereerd. Alle benodigde declaraties en initialisaties zijn al klaar. Ook de (lege) eventhandlers voor de knoppen of andere interface-elementen kunt je laten genereren. Het enige was je als programmeer zelf moet doen is het invullen van de nog lege eventhandlers.

We bekijken een voorbeeld van een eenvoudig Delphi-programma. Je ziet het scherm van dit programma in figuur 6.
Hieronder zie je de code zoals die grotendeels gegenereerd is bij het tekenen van de interface. Bovenaan zie je de declaraties van de verschillende objecten. Dezelfde objecten zijn aanwezig in de object tree view. Door hier, of op het ontwerpscherm een object te

selecteren, kan het betreffende object in de object inspector worden bewerkt. Zo kan een achtergrondkleur gekozen worden, een font, of een tekst voor het opschrift (bijvoorbeeld bij een label). Het uiterlijk van het object wordt meteen aangepast in het ontwerpscherm, dus dit werkt heel makkelijk.
Onderaan in de code zie je de eventhandler voor de optelknop:

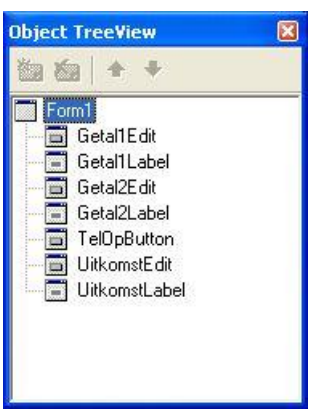
```
procedure Tform1.TelOpButtonClick(Sender: Tobject);
begin
....
end
```

De regels tussen begin en end moeten dus wel door de programmeur zelf ingevuld worden.

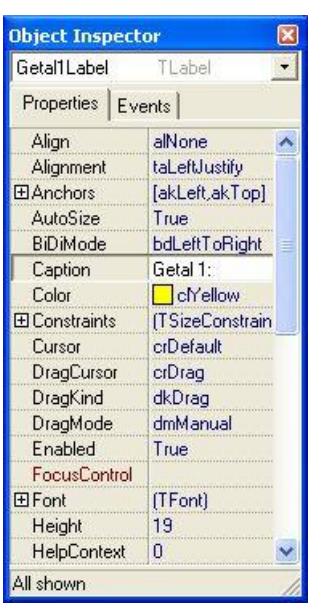


Figuur 6 Codescherm van het programma Optellen.

Figuur 7 Scherm van het programma Optellen



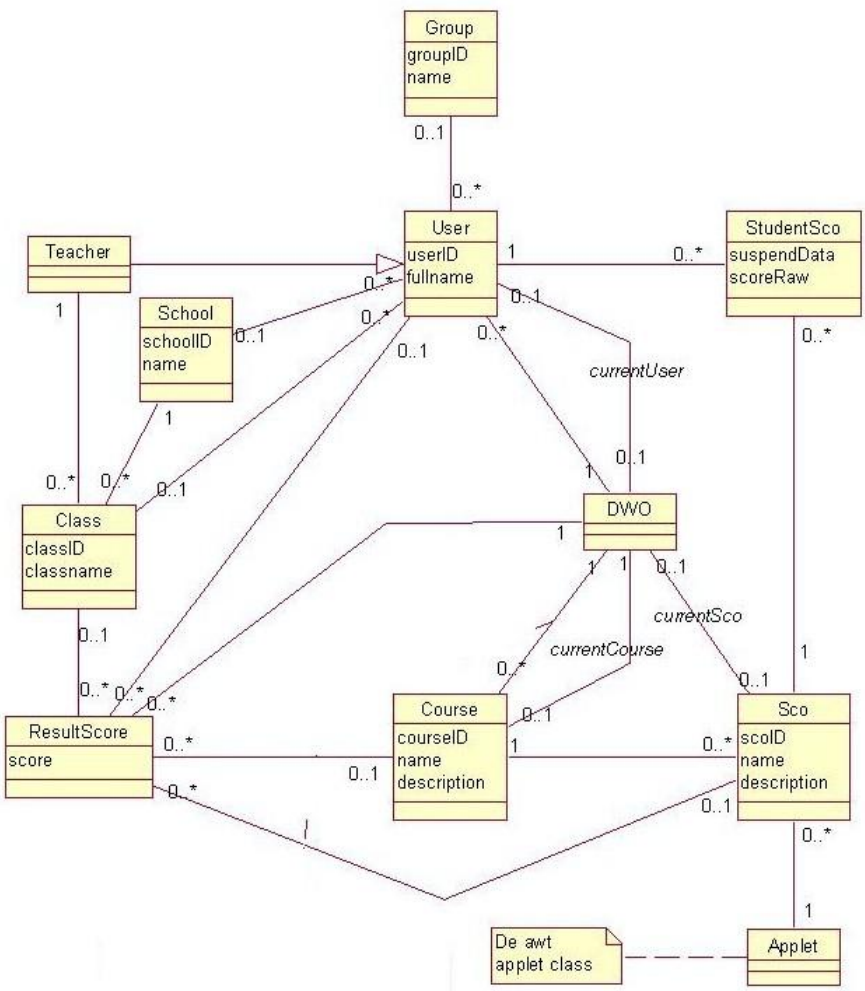
Figuur 8 Object tree view van het programma optellen



Figuur 9 Object inspector van het programma Optellen

9.5 Eigen objecten maken

In de voorbeelden van objectgeorienteerde programma's die we tot nu toe zagen waren objecten eigenlijk steeds (standaard)interface-elementen waarvan de code meegeleverd wordt. Objectgeorienteerd programmeren gaat verder dan dat. Een programmeur die een programma bouwt, maakt niet alleen gebruik van de reeds aanwezige standaardobjecten, maar ontwerpt ook eigen objectsoorten, die hij op verschillende plaatsen in zijn programma kan gebruiken. Het opdelen van een programma in verschillende 'samenwerkende' objecten, die allemaal hun eigen verantwoordelijkheid hebben in het programma is een complexe ontwerpvaardigheid. Het betreft specialistisch werk met eigen tools waarop we hier niet ingaan. Wel aardig om te zien is een onderdeel van een ontwerpdiagram van een complex programma. De 'ontwerptaal' waarin dit diagram gemaakt is heet UML.



Figuur 10 UML-diagram van een onderdeel van een objectgeorienteerd programma