Aangepast cursus Object-georiënteerd Programmeren voor HI-BIN studenten

Jan Van den Bussche

Academiejaar 2002–2003

Boek: Timothy Budd, An introduction to object-oriented programming, second edition. In voldoende exemplaren aanwezig in de bibliotheek.

Leerstof: Je mag de gedeelten over de programmeertalen Smalltalk, Object Pascal, en Objective-C overslaan. Ook mag je hoofdstukken 2, 18, 19, en 21 overslaan.

Exercises: Test je kennis door een aantal "Exercises" op het eind van elk hoofdstuk te maken.

Voorbeeldexamenvragen: Volgen op de volgende pagina's. Het examen zal **schriftelijk en gesloten-boek** verlopen en bestaan uit analoge vragen.

Succes!

1. Stel we hebben reeds een klasse `Persoon` en we willen nu een klasse `Paar` maken. Een instance van klasse `Paar` moet een getrouwd paar personen voorstellen. We leggen deze taak voor aan programmeur X, die als volgt antwoordt:

   > Dit is eenvoudig op te lossen met inheritance. In een getrouwd paar zijn zowel de man als de vrouw personen. We kunnen dus alle operaties en eigenschappen die we al hebben geprogrammeerd in de klasse `Persoon` overerven in de klasse `Paar`.

   Is programmeur X correct? Argumenteer je antwoord.

2. Beschouw het programma-fragment in Figuur 1. (De gebruike programmeertaal is C++ maar het juiste antwoord op deze vraag hangt hier niet van af.) De toekenningsopdracht `y=x` is niet toegelaten. Waarom niet? Is de omgekeerde toekenning wel toegelaten?

3. Java gebruikt een pointer-semantiek voor object variabelen, en gebruikt dynamic method binding. Beschouw nu het programma-fragment in Figuur 2. Wat is de waarde toegekend aan variabele `i`?

4. Beschouw een klasse die een klasse-variabele bevat (`static` in C++ en Java), en bespreek de verschillen tussen de drie situaties waarin deze variabele private, protected, of public zou zijn.

```
class Persoon {
  public string naam;
};

class Werknemer : public Persoon {
  public int salaris;
};

...

{
  Persoon x;
  Werknemer y;

  y=x; /* fout */
}
```

Figuur 1: Programma-fragment bij vraag 2.

```
class A {
  public int m()
  {
    return 1;
  }
};

class B extends A {
  public int m()
  {
    return 2;
  }
};

...

{
  int i;
  A x;
  B y=new B();
  x=y;
  i=x.m();
}
```

Figuur 2: Programma-fragment bij vraag 3.

2. In C++ bestaat er een belangrijk verschil tussen het statisch refereren naar een object, en het refereren naar een object via een pointer. Gegeven volgende code, wat zijn de waarden van `f(o)` en `g(&o)`?

```
class One {
public: virtual int value() { return 1; }
}

class Two {
public: virtual int value() { return 2; }
}

int f(One x)
{ return x.value(); }

int g(One *x)
{ return x->value(); }

One o;
```

3. "Inheritance for specification" en "inheritance for construction" zijn in zekere zin direct tegenovergesteld aan elkaar. Verklaar.

4. Leg lijn per lijn uit wat er gebeurd in de volgende code, die de datatypes `string` en `map` van de C++ standard template library gebruikt:

```
typedef map<string, int> stringVector;
typedef map<string, stringVector> graph;

string pendleton("Pueblo");
string phoenix("Phoenix");

graph cityMap;

cityMap[pendleton][phoenix] = 4;
```

5. Leg uit: iterator.

1. Object-georiënteerd programmeren kan zonder een expliciet concept van "class" te hebben, via het concept van "delegation".

   (a) Leg uit hoe dit gaat.

   (b) Hoe kunnen classes gesimuleerd worden met delegation?

   (c) Omgekeerd, hoe zou delegation gesimuleerd kunnen worden met classes?

2. Geef gedetailleerde uitleg bij het volgende code fragment, dat een eenvoudig "generic algorithm" implementeert uit de STL:

```
template<class InputIterator, class T>
InputIterator
  find (InputIterator first, InputIterator last, const T& value)
{
  while (first != last && *first != value)
    ++first;
  return first;
}
```

3. Bespreek het concept van "abstract class" en geef een voorbeeld dat het nut van dit concept illustreert.

4. Onderstel een class `K` met een public data veld `a` (van type `int`) en een default constructor die `a` op 0 zet. Beschouw de volgende bijna identieke code fragmentjes in C++ en Java:

C++	Java
`K x;`	`K x=new K();`
`K y;`	`K y;`
`y=x;`	`y=x;`
`y.a=1`	`y.a=1`

   Hebben ze een identiek effect? (Argumenteer je antwoord.)

1. Gegeven volgende code in C++, wat zijn de waarden van `f(o)` en `g(&o)`?

```
class One {
public: virtual int value() { return 1; }
}

class Two: public One {
public: virtual int value() { return 2; }
}

int f(One x)
{ return x.value(); }

int g(One *x)
{ return x->value(); }

Two o;
```

2. "Inheritance for specification" en "inheritance for construction" zijn in zekere zin direct tegenovergesteld aan elkaar. Verklaar.

3. Leg lijn per lijn uit wat er gebeurt in de volgende code, die de datatypes `string` en `map` van de C++ standard template library gebruikt. (Leg ook de structuur en het gebruik uit van deze datatypes.)

```
typedef map<string, int> stringVector;
typedef map<string, stringVector> graph;

string pendleton("Pueblo");
string phoenix("Phoenix");

graph cityMap;

cityMap[pendleton][phoenix] = 4;
```

4. Leg uit: iterator.

1. **(a)** Leg uit: statische binding versus dynamische binding. Illustreer met een voorbeeldje. **(b)** Leg uit wanneer statische, dan wel dynamische binding zal gebruikt worden in C++. **(c)** Vergelijk dit met wat er gebeurt in Java.

2. **(a)** Leg uit: multiple inheritance. Illustreer met een voorbeeldje. **(b)** Leg uit hoe multiple inheritance wordt gerealiseerd in C+, en hoe in Java. **(c)** Geef ook een voorbeeldje van dubbelzinnigheden die kunnen ontstaan als we onbedachtzaam multiple inheritance gebruiken in C++. **(d)** Kan dit ook gebeuren in Java? (Argumenteer je antwoord.)

3. **(a)** Leg uit: slicing. **(b)** Komt slicing ook voor in Java? (Argumenteer je antwoord.) **(c)** Als bij een assignment `x = y` slicing zal optreden, is dan de omgekeerde assignment `y = x` toegelaten? (Argumenteer je antwoord.)

4. Leg uit: public versus private inheritance. Illustreer met voorbeeldjes.

1. Over visibility. Voor vraagjes (1a–1d), bekijk programma-code 1, en antwoord telkens met een rijtje van getalletjes in stijgende volgorde.

   (a) Welke van de 3 statements in functie-body *fa* zijn toegestaan?

   (b) Welke van de 6 statements in functie-body *fb* zijn toegestaan?

   (c) Idem voor functie-body *fc*.

   (d) Welke van de 19 statements in functie-body *main* zijn toegestaan?

   (e) Bespreek private inheritance in wat meer detail. Leg uit wanneer het nuttig is.

   (f) Is er ook zoiets als protected inheritance? Zo ja, wat is het verschil met private inheritance? Zo neen, leg uit waarom het geen zin heeft.

2. Contrasteer hoe men nadenkt over een computerprogramma en zijn werking:

   (a) in conventioneel programmeren;

   (b) in object-georiënteerd programmeren.

3. Op welke manieren kan men aan het hergebruik van programma-code doen in object-georiënteerd programmeren? Zijn deze manieren uniek aan object-georiënteerd programmeren, of kunnen sommige ook gebruikt worden in conventioneel programmeren?

4. Over replacement en refinement.

   (a) Contrasteer deze twee manieren van overriding.

   (b) Beschouw de Beta-achtige programma-code 2. Geef de output van procedure

   `printBusscheAnchor`,

   en leg uit hoe die output tot stand komt.

   (c) Leg uit hoe je refinement kan verwezenlijken in Java of C++ (naar keuze).

5. Zowel Java als C++ bieden ingebouwde functies aan die toelaten (at run-time) te weten te komen tot welke klasse een bepaald object behoort. Leg uit hoe je dit echter zelf zou kunnen programmeren.

6. Over statisch en dynamisch typeren.

   (a) Contrasteer deze twee manieren van typeren in programmeertalen.

   (b) Leg het gebruik van virtual method tables uit in de implementatie van method calls.

   (c) Leg uit waarom virtual method tables niet erg praktisch zijn in de implementatie van dynamisch getypeerde talen, en leg het alternatief van dispatch tables uit. In welke zin zijn virtual method tables echter efficiënter?

Programma-code 1

```
class A {
public:
  int va1;
  void fa(void);
private:
  int va2;
protected:
  int va3;
};
class B : public A {
public:
  int vb1;
  void fb(void);
private:
  int vb2;
protected:
  int vb3;
};
class C : private A {
public:
  int vc1;
  void fc(void);
private:
  int vc2;
protected:
  int vc3;
};
void A :: fa(void)
{
  va1 = 0; // 1
  va2 = 0; // 2
  va3 = 0; // 3
}
void B :: fb(void)
{
  vb1 = 0; // 1
  vb2 = 0; // 2
  vb3 = 0; // 3
  va1 = 0; // 4
```

```
  va2 = 0; // 5
  va3 = 0; // 6
}

void C :: fb(void)
{
  vc1 = 0; // 1
  vc2 = 0; // 2
  vc3 = 0; // 3
  va1 = 0; // 4
  va2 = 0; // 5
  va3 = 0; // 6
}

A a, *p;
B b, *q;
C c, *r;

void main(void)
{
  a.va1 = 0; // 1
  a.va2 = 0; // 2
  a.va3 = 0; // 3
  b.vb1 = 0; // 4
  b.vb2 = 0; // 5
  b.vb3 = 0; // 6
  b.va1 = 0; // 7
  b.va2 = 0; // 8
  b.va3 = 0; // 9
  c.vc1 = 0; // 10
  c.vc2 = 0; // 11
  c.vc3 = 0; // 12
  c.va1 = 0; // 13
  c.va2 = 0; // 14
  c.va3 = 0; // 15
  p = &b; // 16
  q = &a; // 17
  p = &c; // 18
  r = &a; // 19
}
```

Programma-code 2

```
printAnchor:
{
  print '<A HREF=http:';
  INNER
  print '">';
};

printLUCAnchor : printAnchor
{
  print '//www.luc.ac.be/';
  INNER
};

printBusscheAnchor : printLUCAnchor
{
  print '~vdbuss';
  INNER
};
```

1.

(a) 1, 2, 3

(b) 1, 2, 3, 4, 6

(c) 1, 2, 3, 4, 6

(d) 1, 4, 7, 10, 16

(e) Als B een private subklasse is van A, wordt al het materiaal van A dat overgeërfd wordt door B, automatisch private in B. Bij de meer standaard public inheritance blijft hetgeen dat public is in A dat ook in B, en idem voor protected.

Private inheritance is vooral nuttig bij inheritance for construction.

(f) Jawel. Het verschil met public inheritance is dat nu hetgeen dat public was in A, protected wordt in B.

2. In een conventioneel programma bekijken we de computer als een machine die communiceert met de gebruiker via haar geheugen. Tijdens de werking is de machine bezig dit geheugen te manipuleren. Als er uiteindelijk in het geheugen zit wat we willen, is de taak volbracht. Dit noemen we het "pigeon-hole" model.

In een object-georienteerd programma bekijken we de computer als een afgesloten wereld waarin objecten kunnen leven. Objecten hebben allen een eigen toestand, die ze tijdens de werking van het programma kunnen aanpassen als reactie op communicaties met andere objecten via messages. Het probleem dat moet opgelost worden door de computer wordt als het ware gesimuleerd.

3. De twee meest voor de hand liggende manieren zijn inheritance en composition.

Bij composition gebruiken we een reeds geschreven klasse als data type. We kunnen instanties maken van deze klasse, waar we dan de voorgeschreven messages naar kunnen sturen. Dit lijkt erg op een library in conventioneel programmeren. We hergebruiken code simpelweg omdat we de reeds geschreven methods gewoon kunnen oproepen als messages.

Bij inheritance gaan we de functionaliteit van een reeds geschreven klasse uitbreiden. Hetgeen reeds bestaat erven we gewoon over, en dit is dan het hergebruik van code in dit geval. Dit is typisch voor OO programmeren.

4.

(a) Bij replacement gaan we een method overriden door de functionaliteit volledig te vervangen. Bij refinement gaan we de functionaliteit van de te overriden method uitbreiden, verfijnen, aanpassen.

(b) `printBusscheAnchor` is een refinement van `printLUCAnchor`, die op zijn beurt een refinement is van `printAnchor`. We voeren dus in principe de code van deze laatste uit, maar deze code zal wel verfijnd moeten worden.

We beginnen braaf met `<A HREF=http:` te printen. Dan komen we aan `INNER`, waarmee bedoeld wordt dat we hier de gelegenheid krijgen te verfijnen. In eerste instantie doen we dat door de eerste verfijning, `printLUCAnchor`, uit te voeren. We beginnen hiermee door `//www.luc.ac.be/` te printen, en komen dan ook aan een `INNER`, die aangeeft dat we de verfijning `printBusscheAnchor` moeten uitvoeren. We printen dus `~vdbuss`, en komen weer opnieuw een `INNER` tegen. Echter, er is geen verfijning van `printBusscheAnchor` aangeroepen (er is er trouwens geen), dus deze laatste `INNER` doet niets.

De verfijning van `printAnchor` is hiermee vervolledigd. We gaan nu voort met de uitvoering van het derde een laatste statement, zijnde het printen van `">`.

(c) In Java staat het keyword `super` voor de receiver, maar dan wel bekeken als instance van de parent class. Zijn we dus een method m aan het overriden, kunnen we nog altijd deze method oproepen als `super`.$m()$. Op deze manier kunnen we dus functionaliteit aan m toevoegen, en dat is precies refinement.

In C++ is er geen `super`, maar daar kunnen we gewoon $A :: m()$ schrijven, onderstellende dat A de parent class is. Bovendien is er in C++ ook een vorm van refinement ingebouwd, in dat de constructor van een klasse automatisch de constructor van de parent klasse zal uitvoeren.

5. Een heel eenvoudige manier om dit te doen is om een method *myClass*() van type `String` toe te voegen aan elke klasse. In een willekeurige klasse A implementeren we deze method simpelweg als volgt:

```
return "A"
```

Het is wel belangrijk dat we hier late binding toepassen, dus in C++ moeten we deze method `virtual` declareren en callen met een pointer naar het object waarvan we het dynamisch type willen weten.

6.

(a) In een statisch getypeerde taal moet elke gebruikte variabele gedeclareerd worden met een type. Tijdens de uitvoering van het programma kan de variabele enkel objecten voorstellen van dat type (of, in OO programmeren, ook van subtypes.)

In dynamisch getypeerde talen mogen variabelen zonder meer gebruikt worden, zonder gedeclareerd type. Tijdens de uitvoering kunnen ze dan ook objecten voorstellen van eender welk type.

(b) Dit is een erg efficiënte techniek om late binding te implementeren. Voor elke klasse maken we een tabel waarin voor elke message die kan gestuurd worden naar een object van die klasse, een pointer staat naar de code van de bijhorende method. Deze tabellen kunnen at compile time geconstrueerd worden en toegevoegd aan het resultaat van de compilatie. Als nu tijdens de uitvoering van het programma een object x wordt gemaakt van klasse A, krijgt dit object een pointer mee naar de tabel voor A. Een message $x.m()$, met late binding, wordt dan geïmplementeerd als een call naar de code voor m waarnaar gewezen wordt in de tabel geassociëerd met x. Dit is dus heel efficiënt.

(c) In een dynamisch getypeerde taal zijn we, bij een message $x.m()$, er echter helemaal niet zeker van of er zowiezo wel een entry voor m zal staan in de virtual method table geassociëerd met x. Dit moet dus at run time opgezocht worden (de compiler genereert code hiervoor). Dit opzoeken at run time is duidelijk minder efficiënt dan de enkele call die we hadden in puntje (b). Verder, nu we toch at run time moeten opzoeken, is het niet meer nuttig om entries voor *alle* mogelijke

methods in de tabel te hebben. Het is nu even goed om enkel pointers te hebben voor de methods van de klasse zelf, en we voegen er een pointer bij naar de tabel voor de superklasse, waarin het zoeken kan verdergaan moesten we het in de klasse zelf niet vinden. Dit is nu precies het mechanisme dat men dispatch tables noemt.