IV OVERERVING

1 Wat is overerving?

We hebben reeds geleerd hoe inkapseling het mogelijk maakt goed gedefinieerde, opzichzelfstaande objecten te schrijven. Inkapseling maakt het een object mogelijk een ander object te *gebruiken* via berichten. Het *gebruiken* is maar een van de manieren waarop objecten in OOP met elkaar in verband kunnen staan. OOP biedt ook nog een tweede manier waarop objecten met elkaar in verband kunnen staan: *overerving*.

Overerving maakt het u mogelijk de definitie van een nieuwe klasse op de definitie van een al bestaande klasse te baseren. Als u een klasse op een andere klasse baseert, dan zal de definitie van de nieuwe klasse automatisch alle eigenschappen, gedragingen en implementaties erven die in de bestaande klasse aanwezig zijn.

Overerving is een mechanisme waarmee u de definitie van een nieuwe klasse op een bestaande klasse kunt baseren. Het gebruik van overerving betekent dat uw nieuwe klasse alle eigenschappen en gedragingen zal erven die in de bestaande klasse aanwezig zijn. Alle methoden en eigenschappen die in de bestaande klasse voorkomen zullen automatisch ook in de interface van de nieuwe klasse bestaan als een klasse van een andere klasse erft.

Kijk eens naar de volgende klasse:

Employee
- firstName : String - lastName : String
+ Employee(first : String, last : String) + getFirstName( ) : String + setFirstName(mfirstName : String): void + getLastName( ) : String + setLastName(mlastName : String): void + toString( ) : String

met volgende pseudo-codes:

Employee(I: first, last : String)

Type:	constructor
Preconditie:	gedefinieerd zijn in een klasse met dezelfde naam
Postconditie:	Het attribuut firstName krijgt de waarde van het argument first en het atribuut lastName krijgt de waarde van het argument last.
Gebruikt:	setFirstName( ), setLastName( )
Gegevens:	/

```
BEGIN
        setFirstName(first)
        setLastName(last)
EINDE
```

getFirstName(I: /
U: first2: String)

Type: accessor
Preconditie: De eigenschap firstName bestaat.
Postconditie: De inhoud van de eigenschap firstName wordt geretourneerd.
Gebruikt:

```
BEGIN
    first2=firstName
EINDE
```

getLastName( ) is analoog.

setFirstName(I: mfirstName: String
U: /)

Type: mutator
Preconditie: De eigenschap firstName bestaat.
Postconditie: De eigenschap firstName krijgt de waarde van het argument van de methode, nl. mfirstName.
Gebruikt:

```
BEGIN
    firstName = mfirstName
EINDE
```

setLastName( ) is analoog.

toString(I: /
U:name2:String)

Type: methode
Preconditie:
Postconditie: Deze methode retourneert een String die de voornaam en de achternaam achter elkaar bevat.
Gebruikt: getFirstName( ), getLastName( )
Gegevens: /

```
BEGIN
    name2 = getFirstName( ) + ' '+ getLastName( )
EINDE
```

Instanties van een klasse zoals Employee kunnen bijvoorbeeld in een databasetoepassing voor loonlijsten voorkomen. Stel nu dat u een werknemer moet modelleren die op basis van commissie werkt. Een dergelijke werknemer heeft een basisloon, plus een kleine commissie per verkoop. De CommissionWorker is, afgezien van deze eenvoudige vereiste, verder precies hetzelfde als een Employee. Een CommissionWorker is tenslotte een Employee.

Er zijn twee manieren waarop u de nieuwe klasse CommissionWorker kunt schrijven met gewone inkapseling. U zou de code uit Employee gewoon kunnen herhalen en code voor het bijhouden van commissies en het berekenen van het loon kunnen toevoegen. Zou u dat doen, dan zou u echter twee afzonderlijken, maar soortgelijke stukken basiscode moeten onderhouden. Zou u een fout moeten verhelpen, dan zou u dat op beide plaatsen moeten doen.

Het gewoon kopiëren en plakken van de code is dus niet echt een optie. U zult iets anders moeten proberen. U zou een werknemervariabele in de klasse CommissionWorker kunnen opnemen en alle berichten zoals getLastName( ) en getFirstName( ) aan de instantie van Employee kunnen delegeren.

Delegeren is het proces waarbij een object een bericht aan een ander object doorgeeft om aan de een of andere aanvraag te voldoen.

Het delegeren dwingt u echter nog steeds alle methoden in de interface van Employee te herdefiniëren om alle berichten te kunnen doorgeven. Geen van deze beide opties lijkt dus echt te voldoen.

Beschouw de volgende oplossing:

CommissionWorker <erft van Employee>
- salary : reëel getal - commission : reëel getal - quantity : geheel getal
+ CommissionWorker(first : String, last : String, csalary : reëel getal, ccommission : reëel getal, cquantity : geheel getal) + setSalary(weeklySalary : reëel getal):void + setCommission(itemCommission : reëel getal):void + setQuantity(totalSold : geheel getal):void + getSalary( ): reëel getal + getCommission( ): reëel getal + getQuantity( ): geheel getal + earnings( ):reëel getal + toString( ): String

hierbij stelt salary het basis wekelijks salaris voor; commission is de commissie per eenheid en quantity is het aantal verkochte eenheden.

De pseudo-codes:

CommissionWorker(I: first, last : String
csalary, ccommission : reële getallen
cquantity : geheel getal)

Type:	constructor
Preconditie:	gedefinieerd zijn in een klasse met dezelfde naam
Postconditie:	Het attribuut firstName krijgt de waarde van het argument first en het atribuut lastName krijgt de waarde van het argument last. De eigenschappen salary, commission en quantity, krijgen de waarden

van de overeenkomstige argumenten van de constructor
Gebruikt: setSalary( ), setCommission( ), setQuantity( ), de constructor van de klasse Employee.
Gegevens: /

```
BEGIN
    super( first, last) //hier wordt de constructor aangeroepen van de
                                                    //klasse Employee
    setSalary(csalary)
    setCommission(ccommission)
    setQuantity(cquantity)
EINDE
```

setSalary (I: weeklySalary: reëel getal)
U:/)

Type: mutator
Preconditie: In de klasse bestaat een eigenschap die het salaris voorstelt; nl. salary.
Postconditie: Aan de eigenschap salary wordt de waarde toegewezen, dat de invoervariabele weeklySalary van deze mutator bevat.
Gebruikt: /
Gegevens: /

```
BEGIN
    ALS(weeklySalary > 0)
        DAN
            salary = weeklySalary
        ANDERS
            salary = 0
    EINDE-ALS-DAN
EINDE
```

setCommission (I: itemCommission: reëel getal)
U:/)

Type: mutator
Preconditie: In de klasse bestaat een eigenschap die de commissie voorstelt; nl. commission.
Postconditie: Aan de eigenschap commission wordt de waarde toegewezen, dat de invoervariabele itemCommission van deze mutator bevat.
Gebruikt: /
Gegevens: /

```
BEGIN
    ALS(itemCommission > 0)
        DAN
            commission = itemCommission
        ANDERS
            commission = 0
```

```
            EINDE-ALS-DAN
        EINDE
```

setQuantity(I: totalSold : geheel getal
U:/)

Type:	mutator
Preconditie:	In de klasse bestaat een eigenschap die de verkochte hoeveelheid voorstelt; nl. quantity.
Postconditie:	Aan de eigenschap quantity wordt de waarde toegewezen, dat de invoervariabele totalSold van deze mutator bevat.
Gebruikt:	/
Gegevens:	/

```
        BEGIN
            ALS(totalSold > 0)
                DAN
                    quantity = totalSold
                ANDERS
                    quantity = 0
            EINDE-ALS-DAN
        EINDE
```

De accessors kan je zelf schrijven.

earnings(I: /
U: earnings2: reëel getal)

Type:	methode
Preconditie:	/
Postconditie:	De totale wekelijkse verdienste van een CommissionWorker wordt berekend en geretourneerd.
Gebruikt:	getSalary( ), getCommission( ), getQuantity( )

```
        BEGIN
            earnings2=getSalary( )+getCommission( )*getQuantity( )
        EINDE
```

toString(I: /
U:name2:String)

Type:	methode
Preconditie:	
Postconditie:	Deze methode retourneert een String die het soort werknemer, de voornaam en de achternaam, van deze werknemer, achter elkaar bevat.
Gebruikt:	De methode toString( ) van de klasse Employee
Gegevens:	/

```
        BEGIN
            name2 = “Commission worker” + super.toString( )
        EINDE
```

Merk op dat om aan te duiden dat toString uit de superklasse komt, wordt het

voorafgegaan door het sleutelwoord super (hierover later nog meer uitleg).

De definitie van CommissionWorker is hier gebaseerd op de bestaande klasse Employee. CommissionWorker erft van Employee : toString( ), getFirstName ( ), getLastName ( ), setFirstName ( ), setLastName ( ), firstName en lastName maken daarom allemaal deel uit van de definitie.

De publieke interface van Employee wordt een onderdeel van de interface van CommissionWorker. U kunt daarom elk bericht naar CommissionWorker sturen dat u ook naar Employee zou kunnen sturen.

Kijk eens naar de volgende main( ), die precies dat doet.

Test( ):

Type:	main-functie
Preconditie:	De klassen Employee en CommissionWorker bestaan.
Postconditie:	Er wordt een CommissionWorker gemaakt. De methodes waarover deze klasse beschikt worden uitgetest.
Gebruikt:	De klasse CommissionWorker
Gegevens:	c : CommissionWorker

```
BEGIN
    // maak een nieuwe CommissionWorker
    c = nieuw CommissionWorker("Mr.", "Sales", 5.50, 1.00,1)
    c.setQuantity(5)

    VOERUIT(Scherm," First Name: ", c.getFirstName( ), N_R)
    VOERUIT(Scherm, "Last Name: ", c.getLastName( ) ,N_R)
    VOERUIT(Scherm, "Total Pay: $ ", c.earnings( ))

EINDE
```

Hieronder wordt het resultaat getoond van deze code

```
First Name: Mr
Last Name: Sales
Total Pay: $10.5
```

2 Wat is het doel van overerving?

Zoals u in het vorige voorbeeld zag, is de eenvoudige *gebruiksrelatie* van inkapseling soms gewoon niet genoeg. De overerving omvat echter meer dan alleen gewoon het erven van een publieke interface en een implementatie.

Zoals u in dit voorbeeld ziet en ook verder zult zien, maakt overerving het de ervende klasse mogelijk elk willekeurig gedrag te herdefiniëren dat deze klasse niet bevalt. Denk maar aan de methode toString( ). Een dergelijk nuttig kenmerk maakt het u mogelijk uw

software aan te passen als uw eisen veranderen. Moet u een wijziging uitvoeren, dan schrijft u gewoon een nieuwe klasse die de oude functionaliteit erft. U vervangt daarna de functionaliteit die gewijzigd moet worden, of voegt de ontbrekende functionaliteit toe, en u bent al klaar. Het vervangen (of *overriding*) is nuttig omdat u daarmee de manier kunt veranderen waarop een object werkt, zonder dat u daar de originele klassedefinitie voor moet veranderen. U kunt uw goed geteste, gevalideerde basiscode intact laten. Het vervangen werkt zelfs als u de originele broncode van een klasse niet hebt.

De overerving heeft nog een ander heel belangrijk gebruiksdoel. We hebben vroeger reeds gezien hoe een klasse gerelateerde objecten groepeert. Overerving maakt het u mogelijk gerelateerde klassen te groeperen en te classificeren.

3 'Is een' versus 'bevat een': leren wanneer u overerving moet gebruiken

U hebt gezien dat de overerving het uw klassen mogelijk maakt de implementatie van andere klassen te erven. Vandaar dat men spreekt over *overerving van implementatie*. Maar betekent het feit dat de ene klasse van de andere kan erven ook dat deze dat moet doen?

Hoe weet u nu wanneer u overerving moet gebruiken? Er is gelukkig een vuistregel die u kunt volgen om onjuist gebruik van overerving te vermijden.

Denkt u erover na overerving te gebruiken voor hergebruik, of voor welk ander gebruiksdoel dan ook, dan moet u zichzelf telkens eerst vragen of de ervende klasse van hetzelfde type is als de klasse waarvan wordt geërfd. Het denken in termen van het type tijdens het overerven wordt vaak de *'is een'*-test genoemd.

'Is een' beschrijft de relatie waarbij een klasse gezien wordt, zijnde, hetzelfde type als een andere klasse.

U gebruikt de *'is een'*-test door tegen uzelf te zeggen: "Een CommissionWorker *'is een'* Employee." Deze bewering is waar en u weet daarom meteen dat overerving in deze situatie geldig is.

Er zullen veel situaties zijn waarin de *'is een'*-test mislukt als u de een of andere implementatie wilt hergebruiken. Er zijn gelukkig andere manieren voor het hergebruiken van de implementatie. U kunt altijd samenstelling en delegatie gebruiken. De *'bevat een'*-test brengt, in dat geval, redding.

'Bevat een' beschrijft de relatie waarbij de ene klasse een instantie van een andere klasse bevat.

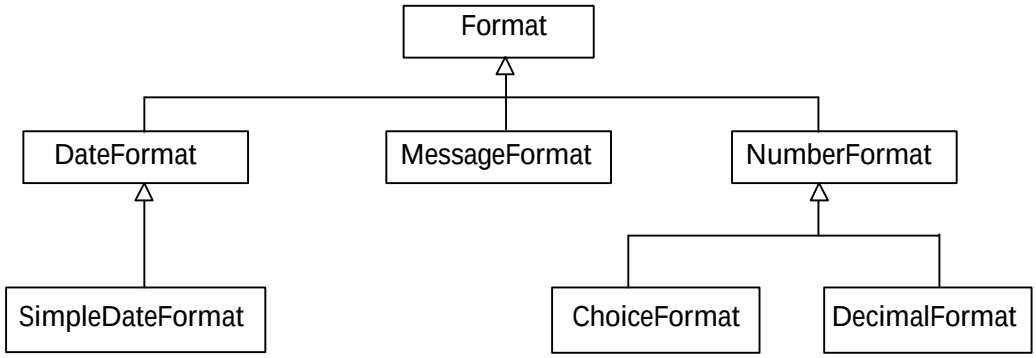
Samenstelling (of composition) betekent dat een klasse geïmplementeerd is met interne variabelen (die lidvariabelen worden genoemd) die instanties van andere klassen bewaren.

4 Leren door de warboel van overervingen te navigeren

De ideeën van ‘is een’ en samenstelling wijzigen de aard van de bespreking van overerving, en wel van een bespreking van het hebzuchtig hergebruiken van implementaties in een bespreking van de onderlinge relaties tussen klassen.

Een klasse die van een andere klasse erft moet op zo'n manier aan die andere klasse gerelateerd zijn dat de resulterende relaties, of overervingshiërarchieën, zinvol zijn.

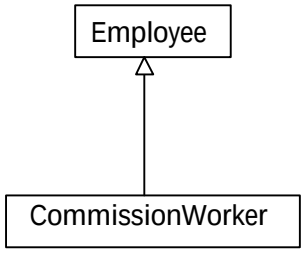
Een *overervingshiërarchie* is een op een boomstructuur lijkend verbindingsdiagram dat de relaties toont die als resultaat van overerving tussen klassen worden gevormd. Figuur 4.1. illustreert een echte, uit Java afkomstige hiërarchie.



Figuur 4.1. Voorbeeldhiërarchie

Overerving definieert de nieuwe klasse, de *child* (het 'kind') in termen van een oude klasse, de *parent* (de 'ouder'). Deze child-parent-relatie is de eenvoudigste overervingsrelatie. Alle overervingshiërarchieën beginnen in feite met een parent en een child.

Op deze manier zijn we gekomen bij de UML-voorstelling van een overervingshiërarchie. Tussen een parent en een child wordt een pijl geplaatst in de zin van de parent. Neem bijvoorbeeld de klasse Employee en CommissionWorker:



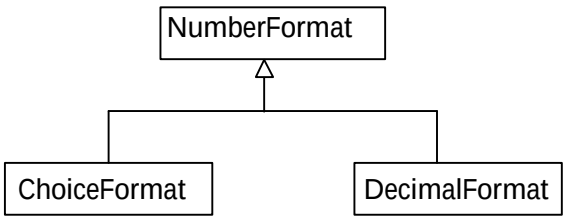
Figuur 4.2.: overervingshiërarchie tussen Employee en CommissionWorker

Als je gebruikt maakt van een diagram dat de overervingshiërarchie weergeeft, zoals in bovenstaande figuur, mag je het stereotype <erft van Employee> weglaten uit de UML-klassevoorstelling van CommissionWorker.

De *child-klasse* is de klasse die erft - deze wordt ook wel de subklasse genoemd.

De *parent-klasse* is de klasse waar de child rechtstreeks van erft - deze wordt ook wel de superklasse genoemd.

Figuur 4.3. illustreert een parent-child-relatie. NumberFormat is de parent van de twee children ChoiceFormat en DecimalFormat.



Figuur 4.3. Een parent met meer children

U kunt de definitie van overerving iets verbeteren, nu u wat meer definities hebt gezien.

Overerving is een mechanisme waarmee u 'is een'-relaties kunt opzetten tussen klassen. Een dergelijke relatie staat het een subklasse ook toe de eigenschappen en gedragingen van zijn superklasse te erven.

Merk op : een child die van een parent erft zal alle eigenschappen en gedragingen krijgen die de parent misschien weer van andere klassen zal hebben geërfd.

Zoals u hebt gezien, moet u alles met de child kunnen doen wat u met de parent ervan kunt doen, wil de overervingshiërarchie zinvol zijn. Dat is waar de 'is een'-test eigenlijk op controleert. Een child mag alleen functionaliteit uitbreiden en toevoegen. Een child mag nooit functionaliteit verwijderen.

Mocht u merken dat een child functionaliteit moet verwijderen, dan geeft dat aan dat de child vóór de parent in de overervingshiërarchie thuishoort!

Parent-klassen en children-klassen zullen net als echte ouders en kinderen op elkaar lijken. Klassen delen type-informatie, in plaats van genen.

Een klasse kan, anders dan bij echte kinderen, maar één fysieke parent hebben. Dat hangt er helemaal van af hoe de taal de overerving implementeert.

Sommige talen staan het een klasse toe meer dan één parent te hebben. Dat staat bekend als *meervoudige overerving* (of *multiple inheritance*).

Andere talen beperken de child tot maar één parent.

Weer andere talen, zoals Java, staan maar één parent toe voor de implementatie, maar leveren een mechanisme voor het erven van meer interfaces (maar met alleen de handtekeningen van de methoden, zonder implementatie).

Child-klassen kunnen net als kinderen van vlees en bloed nieuwe gedragingen en eigenschappen aan zichzelf toevoegen. Een kind van vlees en bloed kan bijvoorbeeld leren piano te spelen, ook al heeft de ouder dat nooit gedaan. Een child kan op een soortgelijke manier een geërfd gedrag herdefiniëren. De ouder kan bijvoorbeeld slecht in wiskunde zijn geweest. Een kind kan extra hard studeren en een goede wiskundestudent worden. Wilt u

een nieuw gedrag aan een klasse toevoegen, dan kunt u dat doen door een nieuwe methode aan de klasse toe te voegen of door een oude methode te herdefiniëren.

4.1. Het mechanisme van de overerving.

Een klasse die van een andere klasse erft, erft de implementatie, gedragingen en eigenschappen(attributen) daarvan. Dat betekent dat alle methoden en attributen die in de interface van de parent beschikbaar zijn ook in de interface van de child zullen voorkomen. Een via overerving geconstrueerde klasse kan drie belangrijke soorten methoden en attributen hebben:

- Vervangen - De nieuwe klasse erft de methode of attribuut van de parent, maar levert een nieuwe definitie.
- Nieuw - De nieuwe klasse voegt een geheel nieuwe methode of attribuut toe.
- Recursief - De nieuwe klasse erft simpelweg een methode of attribuut van de parent.

De meeste OO-talen staan u niet toe een attribuut te vervangen. We bespreken het vervangen van atributen hier echter toch, om redenen van de volledigheid.

Laten we eerst eens een voorbeeld bekijken. We zullen daarna alle soorten methoden en atributen verkennen.

TwoDimensionalPoint
- x_coord : reëel getal - y_coord : reëel getal
+ TwoDimensionalPoint(x: reëel getal, y: reëel getal) + getXCoordinate( ) : reëel getal + setXCoordinate(x : reëel getal) : void + getYCoordinate( ) : reëel getal + setYCoordinate(y : reëel getal) : void + toString( ): String

TwoDimensionalPoint(I: x, y: reële getallen)

Type:	constructor
Preconditie:	gedefinieerd zijn in een klasse met dezelfde naam
Postconditie:	Aan de eigenschappen x_coord en y_coord worden de overeenkomstige waarden van x en y toegewezen.
Gebruikt:	setXCoordinate( ), setYCoordinate( )
Gegevens:	/

```
BEGIN
        setXCoordinate( x )
        setYCoordinate( y )
EINDE
```

setXCoordinate (I: x: reëel getal)
U:/)

Type: mutator
Preconditie: In de klasse bestaat een eigenschap die het x-coördinaat van een tweedimensionaal punt voorstelt, nl. : x_coord.
Postconditie: Aan de eigenschap x_coord wordt de waarde toegewezen, dat de invoervariabele x van deze mutator bevat.
Gebruikt: /
Gegevens: /

```
BEGIN
    x_coord = x
EINDE
```

getXCoordinate(I: /
U: x_coord2: reëel getal)

Type: accessor
Preconditie: De eigenschap x_coord bestaat.
Postconditie: De inhoud van de eigenschap x_coord wordt geretourneerd.
Gebruikt:

```
BEGIN
    x_coord2= x_coord
EINDE
```

setYCoordinate (I: y: reëel getal)
U:/)

Type: mutator
Preconditie: In de klasse bestaat een eigenschap die het y-coördinaat van een tweedimensionaal punt voorstelt, nl. : y_coord.
Postconditie: Aan de eigenschap y_coord wordt de waarde toegewezen, dat de invoervariabele y van deze mutator bevat.
Gebruikt: /
Gegevens: /

```
BEGIN
    y_coord = y
EINDE
```

getYCoordinate(I: /
U: y_coord2: reëel getal)

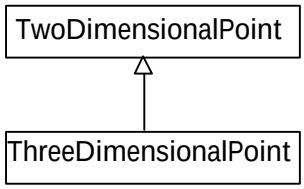
Type: accessor
Preconditie: De eigenschap y_coord bestaat.
Postconditie: De inhoud van de eigenschap y_coord wordt geretourneerd.
Gebruikt:

BEGIN
y_coord2= y_coord
EINDE

toString(I: /
U:name2:String)

Type: methode
Preconditie:
Postconditie: Deze methode retourneert een String die het soort punt , de x-coordinaat en de y-coordinaat, van dit punt, achter elkaar bevat.
Gebruikt: getXCoordinate( ), getYCoordinate( )
Gegevens: /

BEGIN
name2 = "I am a 2 dimensional point.\n" +
"My x coordinate is: "+ getXCoordinate( ) + "\n" +
"My y coordinate is: "+ getYCoordinate( )
EINDE



ThreeDimensionalPoint
- z_coord : reëel getal
+ ThreeDimensionalPoint(x: reëel getal, y: reëel getal, z: reëel getal) + getZCoordinate( ) : reëel getal + setZCoordinate(z : reëel getal) : void + toString( ): String

ThreeDimensionalPoint(I: x, y, z: reële getallen)

Type: constructor
Preconditie: gedefinieerd zijn in een klasse met dezelfde naam
Postconditie: Aan de eigenschappen x_coord , y_coord en z_coord worden de overeenkomstige waarden van x, y en z toegewezen.
Gebruikt: setZCoordinate( ), constructor van de klasse TwoDimensionalPoint
Gegevens: /

BEGIN
super(x, y) // initialiseer de geërfde eigenschappen
// door de constructor van de parent aan te

```
                                        // roepen
                    setZCoordinate(z)
            EINDE


setZCoordinate (    I: z: reëel getal)
                    U:/)
```

Type: mutator
Preconditie: In de klasse bestaat een eigenschap die de z-coördinaat van een driedimensionaal punt voorstelt, nl. : z_coord.
Postconditie: Aan de eigenschap z_coord wordt de waarde toegewezen, dat de invoervariabele z van deze mutator bevat.
Gebruikt: /
Gegevens: /

```
        BEGIN
            z_coord = z
        EINDE
getZCoordinate(   I: /
                  U: z_coord2: reëel getal)
```

Type: Accessor
Preconditie: De eigenschap z_coord bestaat.
Postconditie: De inhoud van de eigenschap z_coord wordt geretourneerd.
Gebruikt:

```
        BEGIN
            z_coord2= z_coord
        EINDE


toString(       I: /
                U:name2:String)
```

Type: methode
Preconditie:
Postconditie: Deze methode retourneert een String die het soort punt , de x-coordinaat , de y-coordinaat en de z-coordinaat, van dit punt, achter elkaar bevat.
Gebruikt: getXCoordinate( ), getYCoordinate( ), getZCoordinate( )
Gegevens: /

```
        BEGIN
              name2 = "I am a 3 dimensional point.\n" +
              "My x coordinate is: "+ getXCoordinate( ) + "\n" +
              "My y coordinate is: "+ getYCoordinate( ) + "\n" +
              "My z coordinate is: "+ getZCoordinate( )
        EINDE
```

U ziet hier twee klassen die geometrische punten representeren, U zou punten kunnen gebruiken in een tekenhulpmiddel, een visueel modelleerprogramma of een vluchtplanner. Er zijn veel praktische gebruiksdoelen voor punten.
TwoDimensionalPoint bevat hier een x- en een y-coördinaat. De klasse definieert methoden voor het ophalen en instellen van de punten en voor het maken van een String-representatie van de puntinstantie.

ThreeDimensionalPoint erft van TwoDimensionalPoint. ThreeDimensionalPoint voegt een z-coördinaat toe en een methode voor het verkrijgen van een String-representatie van de instantie. ThreeDimensionalPoint heeft alle methoden die ook in TwoDimensionalPoint voorkomen, omdat deze van TwoDimensionalPoint erft.

Dit voorbeeld demonstreert alle soorten methoden.

- Vervangen methoden en attributen

Overerving staat u toe een bestaande methode of attribuut te nemen en deze te herdefiniëren. Het herdefiniëren van een methode maakt het u mogelijk het gedrag van het object voor die methode te wijzigen. Een vervangen methode of attribuut zal zowel in de parent als in de child voorkomen. ThreeDimensionalPoint herdefinieert bijvoorbeeld de methode toString ( ) die in TwoDimensionalPoint voorkomt:

```
// uit TwoDimensionalPoint
toString(     I: /
              U:name2:String)
BEGIN
     name2 = "I am a 2 dimensional point.\n" +
     "My x coordinate is: "+ getXCoordinate( ) + "\n" +
     "My y coordinate is: "+ getYCoordinate( )
EINDE
```

TwoDimensionalPoint definieert een methode toString ( ) die de instantie als een tweedimensionaal punt identificeert en de uit twee onderdelen bestaande coördinaat daarvan afdrukt.
ThreeDimensionalPoint herdefinieert de methode toString ( ), zodat deze de instantie als een driedimensionaal punt identificeert en de uit drie onderdelen bestaande coördinaat daarvan afdrukt:

```
// uit ThreeDimensionalPoint
toString(     I: /
              U:name2:String)

BEGIN
     name2 = "I am a 3 dimensional point.\n" +
     "My x coordinate is: "+ getXCoordinate( ) + "\n" +
     "My y coordinate is: "+ getYCoordinate( ) + "\n" +
     "My z coordinate is: "+ getZCoordinate( )
EINDE
```

Kijk nu eens naar de volgende main( ) :

Test( ):

Type:	main-functie
Preconditie:	De klassen TwoDimensionalPoint en ThreeDimensionalPoint bestaan.
Postconditie:	Er worden een TwoDimensionalPoint en een ThreeDimensionalPoint gemaakt en van beiden wordt een string representatie gevraagd.
Gebruikt:	De klassen TwoDimensionalPoint en ThreeDimensionalPoint
Gegevens:	two : TwoDimensionalPoint three : ThreeDimensionalPoint

```
BEGIN
    two = nieuw TwoDimensionalPoint(1, 2)
    three = nieuw ThreeDimensionalPoint(1, 2, 3)
    VOERUIT(Scherm, two.toString( ) )
    VOERUIT(Scherm, three.toString( ) )
EINDE
```

Voer deze main( ) uit en u zal zien dat ThreeDimensionalPoint de vervangen String-representatie levert:

```
I am a 2 dimensional point.
My x coordinate is: 1
My y coordinate is: 2
I am a 3 dimensional point.
My x coordinate is: 1
My y coordinate is: 2
My z coordinate is: 3
```

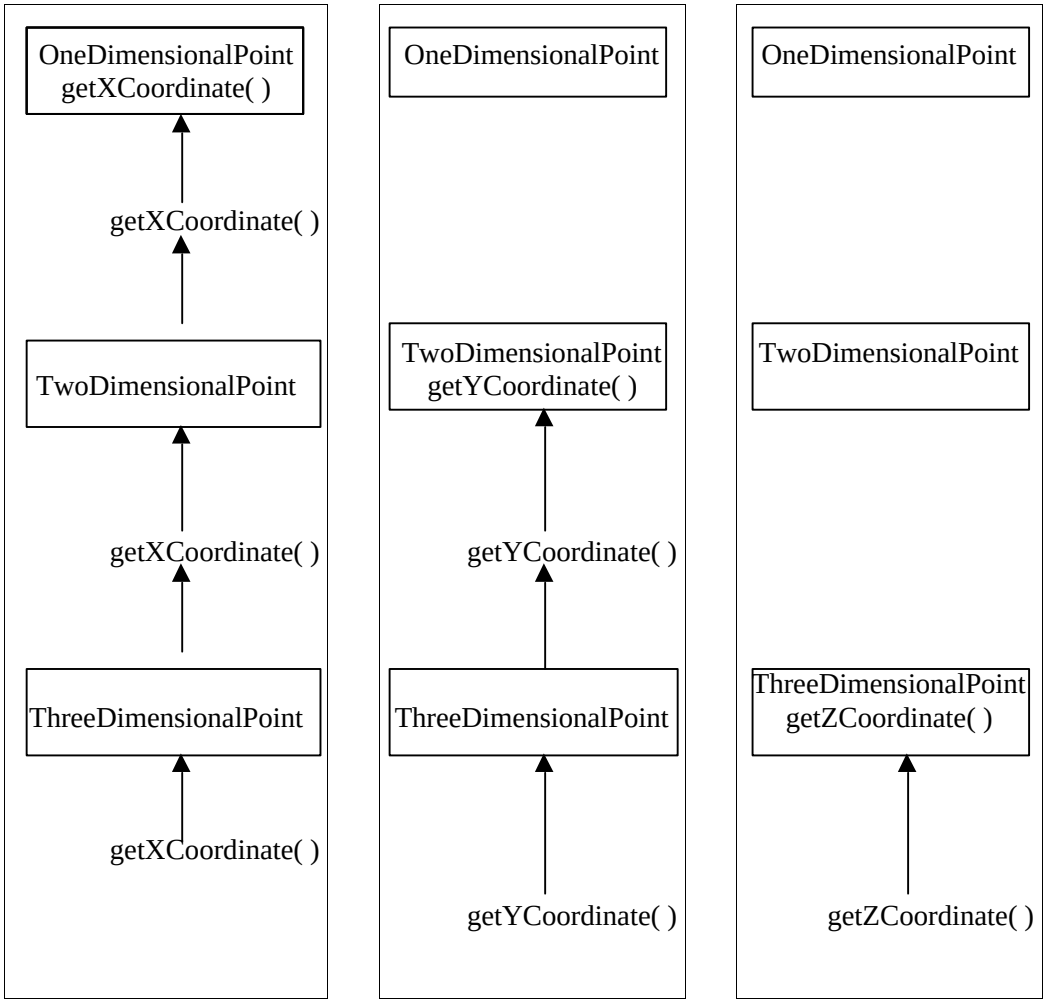
Het vervangen van een methode wordt ook wel het *herdefiniëren* van een methode genoemd. De child levert een aan zijn eigen eisen aangepaste implementatie van de methode door deze te herdefiniëren. Deze nieuwe implementatie levert een nieuw gedrag voor de methode. ThreeDimensionalPoint herdefinieert hier het gedrag van de methode toString( ), zodat deze op de juiste manier naar een String wordt vertaald.

Definitie : Het *vervangen (of overriding)* is het proces van een child die een methode neemt die in de parent voorkomt en die deze herschrijft om het gedrag van de methode te wijzigen. Het vervangen van een methode wordt ook wel het *herdefiniëren* van de methode genoemd.

Maar hoe weet het object nu welke definitie het moet gebruiken?

Het antwoord is afhankelijk van de onderliggende OO-implementatie. De meeste OO-systemen zullen eerst in het object waar het bericht aan is doorgegeven naar de definitie zoeken. Is daar geen definitie te vinden, dan zal de runtime zich door de hiërarchie omhoogwerken tot er een definitie wordt gevonden. Het is belangrijk u te realiseren dat dit de manier is waarop een bericht wordt verwerkt en dat dit de reden is waarom overerving werkt. De definitie van de child zal aangeroepen worden omdat dit de eerste definitie is die wordt gevonden. Het mechanisme is hetzelfde als voor recursieve methoden en eigenschappen, die we later zullen bespreken.

Figuur 4.4. illustreert het doorgeven van de methode tussen de puntobjecten voor een aanroep naar getXCoordinate ( ). Een methodeaanroep naar getXCoordinate ( ) zal door de hiërarchie omhoog worden doorgegeven tot er een definitie voor de methode wordt gevonden.



Figuur 4.4. Het doorgeven van berichten tussen de puntobjecten

Wilt u een methode of een attribuut vervangen, dan moet u zich realiseren, dat niet alle methoden en attributen door uw child kunnen vervangen, of zelfs maar worden gebruikt. De meeste objectgeoriënteerde talen hebben een bepaald niveau aan toegangscontrole. Sleutelwoorden voor de toegangscontrole definiëren wie het precies moet worden toegestaan methoden en attributen te zien en daar toegang toe te krijgen. Deze toegangsniveaus vallen over het algemeen in drie categorieën, die al kort in het begin van dit hoofdstuk werden besproken:

- Private - Privé, een toegangsniveau dat de toegang tot alleen de klasse zelf beperkt.
- Protected - Beveiligd, een toegangsniveau dat de toegang tot de klasse en children daarvan beperkt.

- Public - Publiek, een toegangsniveau dat overal en aan iedereen toegang toestaat.

Beveiligde methoden en attributen zijn de methoden en attributen waartoe u alleen subklassen toegang wilt geven. Laat dergelijke methoden niet publiek. Alleen zij die een uitgebreide kennis van de klasse hebben mogen beveiligde methoden en attributen gebruiken.

U moet alle attributen die geen constanten zijn en alle methoden die alleen voor de klasse zelf bedoeld zijn privé maken. Privé betekent dat geen enkel ander object dan het object zelf de methode kan aanroepen. Maak privé-methoden niet beveiligd alleen omdat de een of andere subklasse er misschien ooit toegang toe zal willen hebben. Gebruik beveiligd alleen voor die methoden waarvan u *weet* dat een subklasse deze wil gebruiken. Gebruik anders privé of publiek. Een dergelijke strenge werkwijze betekent dat u later misschien naar uw code terug zult moeten gaan en het toegangsniveau van een methode zult moeten wijzigen. Deze werkwijze leidt echter tot een strakker ontwerp dan een werkwijze die alles voor een subklasse openstelt.

Opmerking:

Het zal misschien een slechte werkwijze lijken terug te gaan en toegangsniveaus te wijzigen. Overervingshiërarchieën mogen echter nooit per ongeluk optreden. De hiërarchieën moeten zich in plaats daarvan op een natuurlijke manier ontwikkelen terwijl u programmeert. U hoeft zich er niet voor te schamen uw hiërarchieën na verloop van tijd te moeten bijwerken. Echte objectgeoriënteerde programma is een stapsgewijs proces. Denk er echter aan dat de vuistregel is alles privé te maken. Er zijn gevallen waarin dat advies u geen voordeel zal opleveren. Het hangt er feitelijk helemaal van af wat u aan het programmeren bent. Mocht u bijvoorbeeld bibliotheken met algemene klassen verkopen, dan zult u waarschijnlijk standaard beveiligd moeten gebruiken, zodat uw klanten overerving kunnen gebruiken voor het uitbreiden van uw klassen. Er zijn in feite gevallen waarin u een subklasse zult willen ontwerpen met overerving in gedachten. Het is in een dergelijk geval zinvol een overervingsprotocol op te zetten.
Een overervingsprotocol is een abstracte structuur die alleen zichtbaar is via de beveiligde elementen van de klasse. De parent-klasse zal deze methoden aanroepen en de child-klasse kan deze methoden vervangen om het gedrag uit te breiden. U zult daar later een voorbeeld van zien.

Deze definities en regels maken het makkelijk te zien dat beveiligde en publieke methoden en attributen het belangrijkst zijn voor de overerving.

- Nieuwe methoden en attributen

Een nieuwe methode of attribuut is een methode of attribuut die in de child voorkomt, maar niet in de parent. De child voegt de nieuwe methode of attribuut aan zijn interface toe. U hebt nieuwe methoden gezien in het voorbeeld met ThreeDimensionalPoint. ThreeDimensionalPoint voegt de nieuwe methoden getZCoordinate( ) en setZCoordinate( ) toe. U kunt nieuwe functionaliteit aan de interface van uw child toevoegen door nieuwe methoden en attributen toe te voegen.

- Recursieve methoden en attributen

Een recursieve methode of attribuut wordt gedefinieerd in de parent of een andere voorouder, maar niet in de child. Het bericht wordt door de hiërarchie omhoog doorgegeven tot er een definitie van de methode wordt gevonden als u deze methode of attribuut aanspreekt. Het mechanisme is hetzelfde als het in de paragraaf over vervangen methoden en attributen geïntroduceerde mechanisme.

U hebt recursieve methoden gezien in de broncode van TwoDimensionalPoint en ThreeDimensionalPoint. getXCoordinate( ) is een voorbeeld van een recursieve methode, want deze wordt door TwoDimensionalPoint gedefinieerd, maar niet door ThreeDimensionalPoint.

Vervangen methoden kunnen zich ook recursief gedragen. De child zal weliswaar de vervangen methode bevatten, maar de meeste objectgeoriënteerde talen leveren een mechanisme waarmee een vervangen methode de versie van de methode van de parent(of een andere voorouder) kan aanroepen. Dat maakt het u mogelijk de versie van de superklasse te gebruiken terwijl u een nieuw gedrag in de subklasse definieert. Het sleutelwoord super geeft u in Java toegang tot de implementatie van een parent. U zult super gebruiken in de oefeningen.

Merk op dat niet alle talen het sleutelwoord super leveren. U zult voor talen die dit sleutelwoord niet leveren voorzichtig moeten zijn en alle geërfde code op de juiste manier moeten initialiseren. Het niet op de juiste manier verwijzen naar de geërfde klassen kan een subtiele bron van fouten zijn.

5 Soorten overerving

Er zijn in totaal drie manieren waarop u overerving kunt gebruiken:

1. voor het hergebruiken van implementatie;
2. voor het verschil;
3. voor typevervanging.

Wees gewaarschuwd, sommige soorten hergebruik zijn meer wenselijk dan andere! Laten we elk van deze gebruiksdoelen eens in detail bekijken.

5.1. Overerving voor implementatie.

U hebt al gezien dat overerving het een nieuwe klasse toestaat implementatie van een andere klasse te hergebruiken. Het is niet nodig code te knippen en te plakken of een onderdeel te instantiëren en te gebruiken via samenstelling. De overerving maakt de code in plaats daarvan automatisch beschikbaar als een deel van de nieuwe klasse. Uw nieuwe klasse wordt op magische wijze compleet met functionaliteit geboren.

De hiërarchie Employee demonstreert het hergebruiken van implementatie. De child hergebruikt in dit geval een aantal van de gedragingen van de parent.

Tip: Denk eraan dat u aan de implementatie die u erft vastzit als u met overerving van implementatie programmeert. Kies de klasse waar u van erft nauwgezet. U moet de voordelen van het hergebruiken afwegen tegen de eventuele negatieve effecten van het werkelijk hergebruiken van sommige implementaties. Een klasse die op de juiste

manier is gedefinieerd voor overerving zal echter 'zwaar' gebruik maken van fijnmazige, beveiligde methoden. Een ervende klasse kan deze beveiligde methoden vervangen om de implementatie te wijzigen. Het vervangen kan de invloed van de overerving op een slechte of ontoepasselijke implementatie verminderen.

- Problemen van overerving voor implementatie

Het erven van implementatie ziet er tot dusver prachtig uit. Pas echter op: wat op het eerste gezicht een nuttige techniek kan lijken, kan tijdens het gebruik een gevaarlijke werkwijze blijken te zijn. Het erven van implementatie is in feite de zwakste vorm van overerving en u moet deze normaliter vermijden.

Het hergebruik zal misschien makkelijk zijn, maar heeft zoals u zult zien wel een hoge prijs.

U moet naar de typen kijken om de tekortkomingen te kunnen begrijpen. Als een klasse van een andere klasse erft, dan neemt deze automatisch het type van de geërfde klasse aan. Het op de juiste manier erven van het type moet altijd voorrang hebben tijdens het ontwerpen van klassenhiërarchieën. U zult later zien waarom dat zo is. Neem voor nu echter gewoon aan dat dit de waarheid is.

Opmerking 1: Sommige talen staan het een klasse toe alleen implementatie te erven, zonder de type-informatie. De meeste talen staan echter geen scheiding van interface en implementatie toe terwijl er wordt geërfd. Sommige van de talen die deze scheiding wel toestaan doen dat automatisch. Weer andere, zoals C++, staan de scheiding weliswaar toe, maar vereisen dat de programmeur deze expliciet moet aanvragen. Een dergelijke taal vereist dat de programmeur de scheiding moet ontwerpen en aanvragen tijdens het schrijven van de klasse. Het kan natuurlijk redelijk makkelijk zijn over het feit dat u de implementatie en het type van elkaar moet scheiden heen te kijken als u niet voorzichtig bent.

Opmerking 2: Wij gebruiken hier een eenvoudige definitie van overerving. De bespreking van overerving neemt aan dat overerving zowel de implementatie als de interface omvat als de ene klasse van een andere erft.

Een slechte overerving is 'het monster van Frankenstein' voor het programmeren. Gebruikt u alleen overerving voor het hergebruiken van implementatie, zonder andere overwegingen, dan kunt u vaak met een monster komen te zitten dat is samengesteld uit delen die niet bij elkaar horen.

5.2. Overerving voor het verschil.

U hebt overerving voor het verschil gezien in het voorbeeld met TwoDimensionalPoint en ThreeDimensionalPoint. U hebt het ook gezien in het voorbeeld met Employee.
Het programmeren op het verschil maakt het u mogelijk te programmeren door alleen op te geven hoe een child-klasse van zijn parent-klasse verschilt.

Nieuw begrip: *Programmeren op verschil* betekent dat er een klasse wordt geërfd en dat er alleen code wordt toegevoegd die de nieuwe klasse van de geërfde klasse laat verschillen.

U kunt in het geval van ThreeDimensionalPoint zien dat deze van zijn parent-klasse verschilt doordat er een Z-coördinaat wordt toegevoegd. ThreeDimensionalPoint ondersteunt de Z-coördinaat door twee nieuwe methoden toe te voegen voor het instellen en ophalen van deze eigenschap. U ziet verder dat ThreeDimensionalPoint de methode toString( ) herdefinieert.

Het programmeren op verschil is een krachtig concept. U moet alleen voldoende code toevoegen om het verschil tussen de parent- en de child-klasse te beschrijven. U kunt daardoor stapsgewijs programmeren.

Kleinere, beter beheersbare code leidt tot eenvoudiger ontwerpen. Het programmeren van minder code zou, in elk geval in theorie, ook tot minder fouten moeten leiden. U schrijft dus betere code in minder tijd als u op verschil programmeert. U kunt deze stapsgewijze wijzigingen net als bij overerving van implementatie uitvoeren zonder bestaande code te wijzigen.

U kunt op twee manieren op verschil programmeren via overerving: door nieuwe gedragingen en eigenschappen toe te voegen en door oude gedragingen en eigenschappen te herdefiniëren. Deze gevallen staan beide bekend als specialisatie. Laten we de specialisatie eens nader bekijken.

- Specialisatie

Nieuw begrip: *Specialisatie* is het proces van een child-klasse die zichzelf definieert in termen van hoe deze van zijn parent verschilt. De definitie van de childklasse bevat in feite alleen die elementen die de child van zijn parent laten verschillen.

Een child-klasse specialiseert zich verder dan zijn parent door nieuwe methoden en eigenschappen aan zijn interface toe te voegen en bestaande methoden en eigenschappen te herdefiniëren. Het toevoegen van nieuwe methoden, of het herdefiniëren van bestaande methoden, maakt het de child mogelijk gedragingen te uiten die van die van zijn parent verschillen.

Laat u niet in verwarring brengen door de term specialisatie. Specialisatie maakt het u alleen mogelijk door de child van zijn parent geërfde gedragingen en eigenschappen te herdefiniëren, of daar gedragingen en eigenschappen aan toe te voegen. Hoewel de term dat misschien wel lijkt te suggereren, maakt specialisatie het u niet mogelijk geërfde gedragingen en eigenschappen uit de child te **verwijderen**. Een klasse kan niet selectief erven.

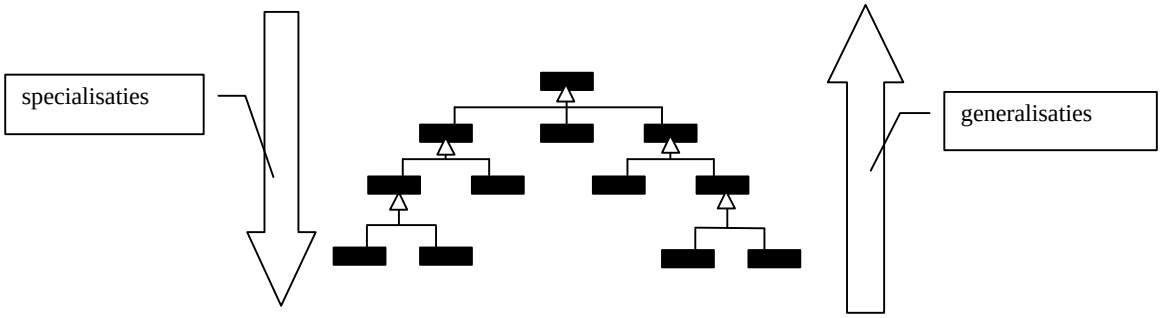
Specialisatie beperkt bijvoorbeeld wat er wel en niet een driedimensionale punt kan zijn. Een ThreeDimensionalPoint kan *altijd* een TwoDimensionalPoint zijn. De stelling dat een TwoDimensionalPoint altijd een ThreeDimensionalPoint kan zijn is echter onjuist.

Een ThreeDimensionalPoint is in plaats daarvan een specialisatie van TwoDimensionalPoint en een TwoDimensionalPoint is een generalisatie van ThreeDimensionalPoint.

Figuur 4.5. illustreert het verschil tussen generalisatie en specialisatie. U specialiseert verder naarmate u door de hiërarchie omlaag beweegt. U generaliseert verder naarmate u door de hiërarchie omhoog beweegt. Er kunnen meer klassen in dezelfde categorie vallen als u generaliseert. Er zullen minder klassen aan alle op dat niveau te categoriseren criteria voldoen als u specialiseert.

Zoals u ziet, betekent specialiseren niet het beperken van de functionaliteit; het betekent het beperken van het aantal typecategorieën.

De specialisatie hoeft niet op te houden bij ThreeDimensionalPoint. De specialisatie hoeft in feite niet eens bij TwoDimensionalPoint te beginnen. Overerving kan zo diep gaan als u maar wilt. U kunt overerving gebruiken voor het vormen van ingewikkelde structuren van klassenhiërarchieën.



Figuur 4.5. : U generaliseert verder naarmate u door een hiërarchie omhoog beweegt. U specialiseert verder naarmate u door een hiërarchie omlaag beweegt.

Het eerder geïntroduceerde idee van hiërarchie leidt tot twee nieuwe termen: voorouder *(ancestor)* en afstammeling *(descendant)*.

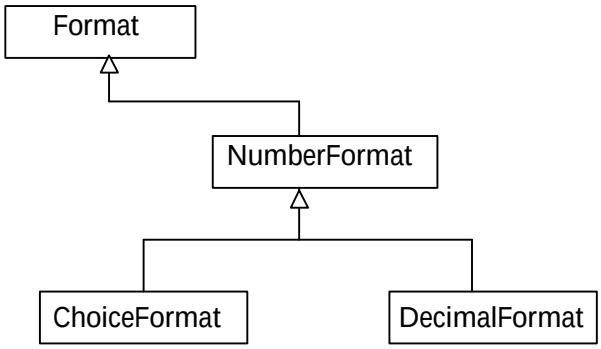
Pas op! Het feit dat u ingewikkelde hiërarchieën kunt hebben, betekent niet dat u die ook moet hebben. U moet ernaar streven ondiepe hiërarchieën te hebben, in plaats van al te diepgaande hiërarchieën. Hiërarchieën worden moeilijker te onderhouden naarmate ze dieper worden.

Een *voorouder* is een klasse die ergens in de klassenhiërarchie voorkomt vóór de parent van een gegeven child. Zoals u in figuur 4.6. kunt zien, is Format een voorouder van DecimalFormat.

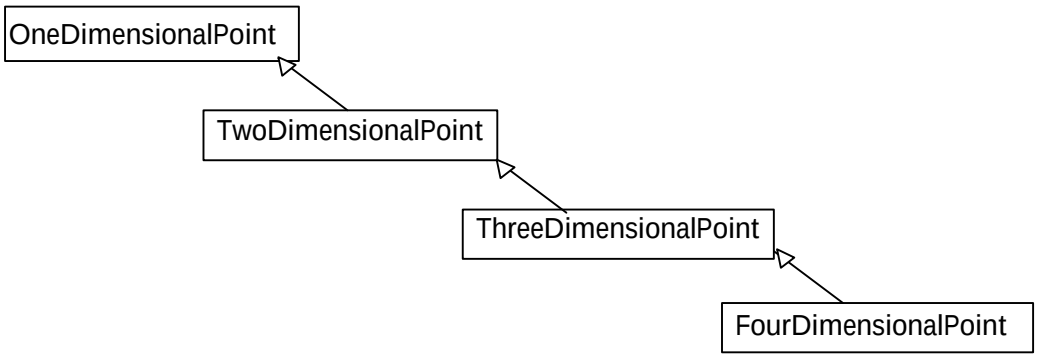
Een klasse die na een andere gegeven klasse in de klassenhiërarchie voorkomt wordt een *afstammeling* van die klasse genoemd. Zoals u in figuur 4.6. ziet, is DecimalFormat een afstammeling van Format.

Stel dat we de in figuur 4.7. getoonde hiërarchie voor het erven van klassen hebben. We zeggen dat OneDimensionalPoint de parent van TwoDimensionalPoint en een voorouder van ThreeDimensionalPoint en FourDimensionalPoint is. We kunnen ook zeggen dat TwoDimensionalPoint, ThreeDimensionalPoint en FourDimensionalPoint

allemaal afstammelingen zijn van OneDimensionalPoint. Alle afstammelingen delen de methoden en eigenschappen van hun voorouders.



Figuur 4.6. : DecimalFormat *een afstammeling van* Format.



Figuur 4.7. : De puntenhiërarchie.

We kunnen nu nog een paar interessante dingen over de klassenhiërarchie zeggen.

OneDimensionalPoint is de hoofdklasse (de *root)* en FourDimensionalPoint is een eindklasse (of een *leaf-klasse).*

Nieuw begrip: De *hoofdklasse* (of *root*) is de bovenste klasse in de overervingshiërarchie. Figuur 4.7. toont dat OneDimensionalPoint een hoofdklasse is.

Nieuw begrip: Een *eindklasse* (of *leaf-klasse*) is een klasse zonder children. U ziet in figuur 4.6. dat DecimalFormat een eindklasse is.

Het is belangrijk op te merken dat de afstammelingen in hun voorouders gemaakte wijzigingen zullen weerspiegelen. Stel dat u een fout vindt in TwoDimensionalPoint. Alle klassen van ThreeDimensionalPoint tot aan FourDimensionalPoint zullen van de wijziging profiteren als u de fout in TwoDimensionalPoint verhelpt. Of u nu een fout verhelpt of een implementatie efficiënter maakt, alle afgestamde klassen in de hiërarchie zullen daar voordeel van hebben.

MEERVOUDIGE OVERERVING

U hebt in de voorbeelden tot dusver alleen enkelvoudige overerving gezien. Sommige implementaties van overerving staan het een enkel object toe rechtstreeks van meer dan één andere klasse te erven. Een dergelijke implementatie van het overerven wordt *meervoudige overerving* genoemd. Meervoudige overerving is een controversieel aspect van het objectgeoriënteerd programmeren. Sommige mensen beweren dat meervoudige overerving software alleen maar moeilijker te begrijpen, te ontwerpen en te onderhouden maakt. Andere mensen zweren erbij en beweren dat een taal zonder meervoudige overerving niet compleet is.
Hoe dan ook, meervoudige overerving kan nuttig zijn, mits er voorzichtig en op de juiste manier gebruik van wordt gemaakt. Meervoudige overerving introduceert een aantal problemen. Een volledige bespreking van de voor- en nadelen van meervoudige overerving valt echter buiten het bestek van deze cursus.

5.3. Overerving voor typevervanging.

De laatste soort van overerving is de overerving voor typevervanging. Typevervanging maakt het u mogelijk voor vervanging geschikte relaties te beschrijven. Maar wat is een voor vervanging geschikte relatie nu eigenlijk?.
Kijk eens naar de klasse Line:

Line
- p1 : TwoDimensionalPoint - p2 : TwoDimensionalPoint
+ Line(cp1: TwoDimensionalPoint, cp2 : TwoDimensionalPoint) + getEndPoint1( ): TwoDimensionalPoint + getEndPoint2( ): TwoDimensionalPoint + getDistance( ): reëel getal + getMidpoint( ): TwoDimensionalPoint

Line(I: cp1, cp2: TwoDimensionalPoint)

Type:	constructor
Preconditie:	gedefinieerd zijn in een klasse met dezelfde naam
Postconditie:	Aan de eigenschappen p1 en p2 worden de overeenkomstige waarden van cp1 en cp2 toegewezen.
Gebruikt:	/
Gegevens:	/

```
BEGIN
        p1 = cp1
        p2 = cp2
EINDE
```

getEndPoint1(I: /
U: endPoint1: TwoDimensionalPoint)

Type: accessor
Preconditie: De eigenschap die het eerste eindpunt voorstelt, bestaat: nl. : p1
Postconditie: De inhoud van de eigenschap p1 wordt geretourneerd.
Gebruikt:

```
BEGIN
    endPoint1 = p1
EINDE
```

getEndPoint2(I: /
U: endPoint2: TwoDimensionalPoint)

Type: accessor
Preconditie: De eigenschap die het tweede eindpunt voorstelt, bestaat: nl. : p2
Postconditie: De inhoud van de eigenschap p2 wordt geretourneerd.
Gebruikt:

```
BEGIN
    endPoint2 = p2
EINDE
```

getDistance(I: /
U: distance: reëel getal)

Type: methode
Preconditie:
Postconditie: De waarde van de lengte van het lijnstuk wordt berekend, en in de variabele distance gestoken en deze waarde wordt dan ook wordt geretourneerd.
Gebruikt: getXCoordinate( ), getYCoordinate( )
Gegevens x, y : reële getallen

```
BEGIN
    x = (p2.getXcoordinate( ) – p1.getXcoordinate( ) )^ 2
    y = (p2.getYcoordinate( ) – p1.getYcoordinate( ) )^ 2
    distance = (x + y)^(1/2)
EINDE
```

getMidpoint (I: /
U: midpoint: TwoDimensionalPoint)

Type: methode
Preconditie:
Postconditie: Het middelpunt wordt bepaald en in midpoint gestoken. midpoint wordt geretourneerd.
Gebruikt: getXCoordinate( ), getYCoordinate( ), constructor van TwoDimensionalPoint
Gegevens new_x, new_y : reële getallen

```
BEGIN
    new_x = (p1.getXcoordinate( ) + p2.getXcoordinate( ) )/ 2
```

```
        new_y = (p1.getYcoordinate( ) + p2.getYcoordinate( ) )/ 2
        midpoint = nieuw TwoDimensionalPoint(new_x , new_y)
EINDE
```

Line neemt twee TwoDimensionalPoints aan als attributen en levert een paar methoden voor het ophalen van de waarden, een methode voor het berekenen van de afstand tussen de punten en een methode voor het berekenen van het middelpunt.

Een voor vervanging geschikte relatie betekent dat u *elk willekeurig* object dat van een TwoDimensionalPoint erft aan de constructor van Line kunt doorgeven.

Denk eraan dat er van een child die van zijn parent erft gezegd kan worden dat de child een parent 'is' (de eerder in dit hoofdstuk beschreven 'is een' relatie). U kunt dus een ThreeDimensionalPoint aan de constructor doorgeven omdat een ThreeDimensionalPoint een TwoDimensionalPoint 'is'.

Kijk eens naar de volgende main( ) :

Test( ):

Type:	main-functie
Preconditie:	De klassen TwoDimensionalPoint, Line en ThreeDimensionalPoint bestaan.
Postconditie:	Er wordt een driedimensionaal punt en een tweedimensionaal punt gemaakt. Er wordt een lijnstuk met die twee eindpunten gemaakt en het middelpunt van deze lijn wordt berekend en afgedrukt.
Gebruikt:	De klassen TwoDimensionalPoint, Line en ThreeDimensionalPoint
Gegevens:	p1 : ThreeDimensionalPoint p2 : TwoDimensionalPoint lijn : Line mid : TwoDimensionalPoint

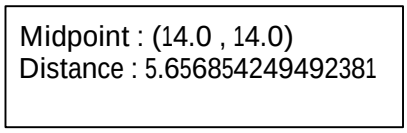
```
BEGIN
        p1 = nieuw ThreeDimensionalPoint(12, 12, 2)
        p2 = nieuw TwoDimensionalPoint(16, 16)

        lijn = nieuw Line( p1, p2)
        mid = lijn.getMidpoint( )

        VOERUIT(Scherm, "Midpoint: (" , mid.getXCoordinate( ) , ", " ,
                        mid.getYCoordinate( ) , ")" )
        VOERUIT(Scherm, "Distance: ",lijn.getDistance( ) )
EINDE
```

U zult opmerken dat de main( ) zowel een TwoDimensionalPoint als een ThreeDimensionalPoint aan de constructor van de lijn doorgeeft. Figuur 4.8. illustreert wat u te zien zult krijgen als u deze main uitvoert.

Opmerking: Probeert u zich eens voor te stellen welke mogelijkheden voor vervanging geschikte relaties u bieden. Voor vervanging geschikte relaties zouden u het in het voorbeeld van de lijn bijvoorbeeld misschien mogelijk kunnen maken in een GUI snel van een driedimensionale naar een tweedimensionale weergave over te schakelen.



Figuur 4.8. : Voor vervanging geschikte relaties uitproberen.

Geschiktheid voor inpluggen *(pluggability)* is een krachtig concept. U kunt elk bericht aan een child sturen dat u ook aan de parent ervan zou kunnen sturen. Dat betekent dat u de child kunt behandelen als zou deze uitwisselbaar zijn met zijn parent. Dat is ook de reden waarom u nooit gedragingen *mag verwijderen* tijdens het maken van een child. Zou u dat doen, dan zou de geschiktheid voor inpluggen wegvallen.

Geschiktheid voor inpluggen betekent dat u op elk gewenst moment nieuwe subtypen aan uw programma kunt toevoegen. Weet uw programma hoe het een voorouder moet gebruiken, dan weet het ook hoe het de nieuwe objecten moet gebruiken. Het programma hoeft zich er niet druk over te maken wat nu eigenlijk precies het type van het object is. Het programma zal het object kunnen gebruiken, zolang dat object maar een voor vervanging geschikte relatie tot het verwachte type heeft.

Pas op: Denk erom dat voor vervanging geschikte relaties maar tot op een bepaald punt omhoog in de overervingshiërarchie kunnen gaan. Hebt u een methode geschreven dat een bepaald type object kan aannemen, dan kunt u daar niet de parent van het verwachte object aan doorgeven. U kunt er echter wel elke willekeurige afstammeling van dat object aan doorgeven.

Neem bijvoorbeeld de constructor van Line:

+ Line(cp1: TwoDimensionalPoint, cp2 : TwoDimensionalPoint)

U kunt een TwoDimensionalPoint of een willekeurige afstammeling van TwoDimensionalPoint aan de constructor doorgeven. U kunt echter geen OneDimensionalPoint aan de constructor doorgeven, want deze staat hoger in de klassenhiërarchie dan TwoDimensionalPoint.

Nieuw begrip: Een *subtype* is een type dat een ander type uitbreidt via overerving.

Geschiktheid voor inpluggen vergroot de kans op hergebruik. Stel dat u een container hebt geschreven voor het bevatten van TwoDimensionalPoint's. U kunt de container wegens de geschiktheid voor inpluggen ook voor elke willekeurige afstammeling van TwoDimensionalPoint gebruiken.

Geschiktheid voor inpluggen is belangrijk, want hiermee kunt u algemene code schrijven. U kunt gewoon objecten schrijven die met objecten van het type TwoDimensionalpoint kunnen omgaan, in plaats van een aantal meervoudige ALS-structuren (case in Java) of

ALS-DAN-controles te moeten schrijven om te zien wat voor soort objecten er momenteel in het programma worden gebruikt.

6 Tips voor effectieve overerving

Overerving heeft zijn eigen reeks ontwerpproblemen. Overerving is weliswaar krachtig, maar biedt u ook de mogelijkheid kuilen te graven waar u zelf in kunt vallen als u overerving niet op de juiste manier gebruikt. De volgende tips zullen u helpen effectief gebruik te maken van overerving:

- Gebruik overerving voornamelijk voor het hergebruiken van interfaces en voor het definiëren van voor vervanging geschikte relaties. U kunt overerving ook gebruiken voor het uitbreiden van een implementatie, maar alleen als de resulterende klasse aan de 'is een'-test voldoet.
- Geef over het algemeen de voorkeur aan samenstelling boven overerving voor eenvoudige gevallen voor het hergebruiken van implementatie. Gebruik alleen overerving als u de 'is een'-test op de resulterende hiërarchie kunt toepassen. Gebruik overerving niet voor het gretig hergebruiken van implementatie.
- Gebruik altijd de 'is een'-regel.

Goede overervingshiërarchieën treden niet uit zichzelf op. U zult vaak hiërarchieën ontdekken terwijl u werkt. Herschrijf uw code als dat gebeurt. U zult uw hiërarchieën soms ook weloverwogen moeten ontwerpen. Er zijn hoe dan ook een paar ontwerpprincipes die u kunt volgen:

- Houd de vuistregel aan dat uw klassenhiërarchieën relatief ondiep moeten zijn.
- Ontwerp uw overervingshiërarchieën nauwgezet en verplaats algemene dingen naar abstracte basisklassen. Abstracte basisklassen maken het u mogelijk een methode te definiëren zonder een implementatie te specificeren. U kunt een abstracte basisklasse niet instantiëren, omdat deze geen implementatie specificeert. Het abstractiemechanisme dwingt een ervende klasse echter een implementatie te leveren. Abstracte klassen zijn nuttig voor geplande overerving. Ze helpen de ontwikkelaar te zien wat hij moet implementeren.

  Opmerking: Mocht uw taal geen abstractiemechanisme leveren, dan kunt u lege methoden maken en documenteren dat die methoden in hun geheel door subklassen moeten worden geïmplementeerd.

- Klassen hebben vaak gemeenschappelijke code. Het heeft geen zin meer kopieën van dezelfde code te hebben. U moet gemeenschappelijke code verwijderen en deze in een enkele parent-klasse isoleren. Verplaats de code echter niet te ver omhoog in de hiërarchie. Verplaats de code alleen naar het eerstvolgende niveau boven het niveau waarop het nodig is.
- U kunt uw hiërarchieën domweg niet altijd in hun geheel plannen. Er zullen u geen gemeenschappelijkheden opvallen, tot u dezelfde code een paar maal hebt geschreven. Zie er niet tegen op uw klassen te herschrijven als u een gemeenschappelijkheid opmerkt. Het op die manier herschrijven wordt vaak ook wel *refactoring* genoemd.

Inkapseling is even belangrijk tussen parent en child als tussen niet-gerelateerde klassen. Pas er voor op niet zorgeloos met de inkapseling om te gaan als u erft. De werkwijze een goed gedefinieerde interface te gebruiken is even geldig tussen een parent en een child als tussen in het geheel niet aan elkaar gerelateerde klassen. Hier volgen een paar tips die u zullen helpen ervoor op te passen de inkapseling te verstoten als u erft:

- Gebruik goed gedefinieerde interfaces tussen de parent en de child, net als u dat tussen klassen zou doen.
- Mocht u methoden toevoegen die specifiek bedoeld zijn voor gebruik door subklassen, dan moet u erom denken ze beveiligd (protected) te maken, zodat alleen de subklasse ze kan zien. Beveiligde methoden maken het u mogelijk uw subklassen wat meer controle te geven, zonder deze controle voor elke klasse beschikbaar te maken.
- U moet over het algemeen vermijden de interne implementatie van uw objecten voor subklassen open te stellen. Een subklasse zou anders afhankelijk van de implementatie kunnen worden.

Hier volgen nog een paar afsluitende tips voor effectieve overerving:

- Vergeet nooit dat vervanging het voornaamste doel is. Zelfs als een object 'intuïtief' in een hiërarchie lijkt thuis te horen, betekent dat nog niet dat het object daar ook werkelijk in thuishoort. Het feit dat u het kunt doen of dat uw intuïtie u vertelt dat u dat moet doen betekent nog niet dat u het ook werkelijk moet doen.
- Programmeer op verschil om uw code hanteerbaar te houden.
- Geef altijd de voorkeur aan samenstelling boven overerving voor het hergebruiken van implementatie. De bij het samenstellen betrokken klassen kunnen over het algemeen makkelijker worden gewijzigd.