De UML-associatie is een concept dat vaak verkeerd begrepen wordt. Het meest voorkomende misverstand is dat een associatie hetzelfde is als een pointer, of misschien twee pointers, in een programma. UML-associaties zijn echter veel krachtiger. Dit artikel zal de betekenis van de UML-associatie verklaren, evenals de wijze van implementatie. De gegeven implementatie-voorbeelden zijn geschreven in Java, maar ze kunnen eenvoudig omgezet worden in een andere programmeertaal.



# Implementatie van UML associaties

## *Concept en toepassingen*

De codevoorbeelden in dit artikel werden allemaal volledig gegenereerd door het UML/OCL-tool Octopus, die kan worden gedownload van http://www.klasse.nl/octopus/index.html. De codevoorbeelden zelf zijn ook beschikbaar via de website van dit blad (www.release.nl).

### Het verschil tussen associaties en pointers

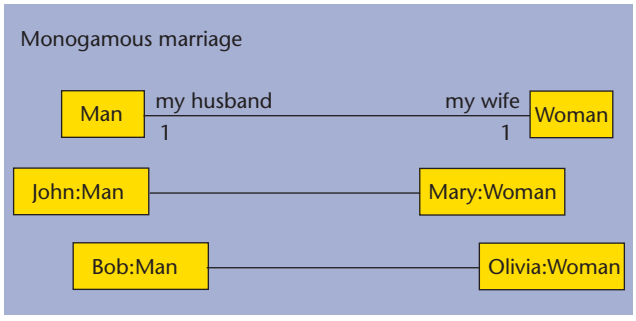
In een programmeertaal kan een pointer worden gebruikt om aan een ander element te refereren. Dat andere element kan een enkel object zijn of een verzameling objecten (bijvoorbeeld een list of set). Het gerefereerde element is zich niet bewust dat sommige objecten naar hem refereren. Verder mag het object dat de referentie bevat, deze referentie vrijelijk distribueren naar andere objecten.

Een pointer is eenvoudigweg informatie over hoe je een element kunt vinden, vergelijkbaar met een website-adres. Als ik het adres zou hebben van de website waarop bijvoorbeeld alle geheimen van de CIA te vinden zijn, dan hoeft de CIA niet op de hoogte zijn dat ik dat adres heb. Bovendien kan ik dat adres met iedereen delen, terwijl de CIA dat nooit te weten hoeft te komen.

Een UML-associatie is, evenals een pointer, een referentie naar een ander element of een verzameling elementen, met dit verschil dat het andere element zich bewust is van de referentie. Door een UML-associatie te gebruiken, zou de CIA weten wie ik ben en dat ik hun adres heb. De meeste associaties zijn wederzijds navigeerbaar. Wanneer object A een referentie naar object B bevat, dan heeft object B per definitie een referentie naar object A. Verderop in dit artikel zullen we one-way navigeerbare associaties bespreken.

### Associaties zijn 'huwelijken'

UML-associaties vertegenwoordigen relaties. De beste manier om een associatie uit te leggen, is dan ook de vergelijking met een huwelijk, zoals weergegeven in figuur 1. Deze figuur toont een class-diagram met de klassen *Man* en *Woman*, en een associatie tussen deze klassen. Ook worden twee voorbeeldinstanties van deze configuratie getoond: een huwelijk tussen John en Mary, en één tussen Bob en Olivia.




**Figuur 1. UML-associaties vertegenwoordigen relaties.**

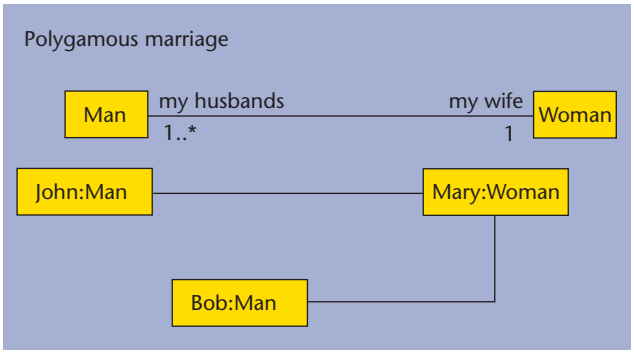
Elk paar objecten dat verbonden is via een associatie, is 'getrouwd', net als John en Mary. Zo'n huwelijk heeft een aantal karakteristieken, die we de ABACUS-regel zullen noemen.

- *Awareness*: beide objecten zijn zich bewust dat de relatie bestaat.
- *Boolean existance*: als de partners instemmen met een scheiding wordt de relatie, in UML genaamd link,

volledig ontbonden.

- *Agreement*: beide objecten moeten instemmen met de relatie; zij moeten zeggen "Ja, ik wil".
- *Cascaded deletion*: als één van de partners komt te overlijden, wordt de link eveneens verbroken.
- *USe of rolenames*: een object mag aan zijn partner refereren door de rolnaam te gebruiken waarin de associatie voorziet: 'my husband' of 'my wife'.

**MULTIPLICITEITEN** Alles goed en wel, maar associaties kunnen multipliciteiten hebben die anders zijn dan de één-op-één versie in een monogaam huwelijk. Wat gebeurt er wanneer één van de associaties meerdere referenties bevat? In dat geval is sprake van een polygaam huwelijk.

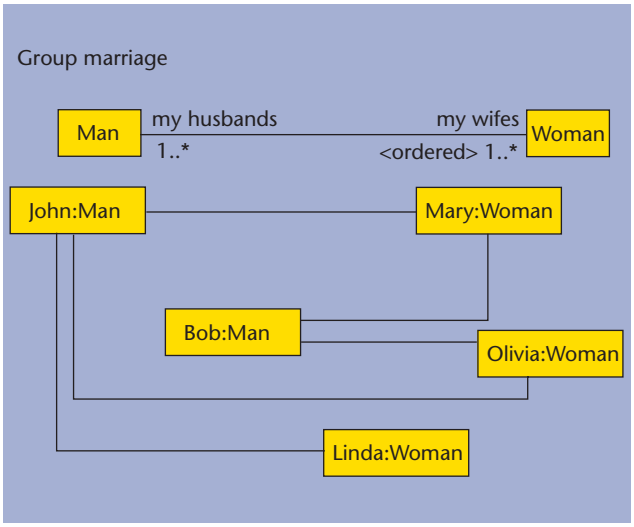



**FIGUUR 2. Associaties kunnen meerdere referenties bevatten.**

Ook hier zijn alle ABACUS-regels voor associaties nog steeds van toepassing. Net als in het monogame huwelijk zijn de objecten aan beide zijden van de relatie zich bewust dat de relatie bestaat. Een object aan de plurale kant hoeft niet op de hoogte zijn van de andere objecten die zich in de relatie bevinden. In dit voorbeeld betekent dat, dat John weet dat hij met Mary getrouwd is. Hij weet dat dit een polygaam huwelijk betreft, maar hij hoeft niet te weten wie Mary's andere echtgenoten zijn. Datzelfde geldt voor Bob. Mary daarentegen kent al haar echtgenoten, zowel Bob als John.

Eveneens kunnen Bob en John aan Mary refereren als 'my wife', Mary mag aan de groep (verzameling) bestaande uit John en Bob refereren als 'my husbands'. Het refereren aan een enkel element in deze groep, bijvoorbeeld aan Bob, wordt echter lastiger. Mary moet een selectiemechanisme gebruiken voor haar verzameling echtgenoten, bijvoorbeeld de naam van de echtgenoot. Geschreven in OCL, zou Mary een expressie als *self.myHusbands->select(name = 'Bob')* kunnen gebruiken. Als Mary en Bob besluiten om te scheiden, dan is de link tussen beiden ontbonden. De link tussen Mary en John blijft.

Het geval waar beide associatie-einden een multipliciteit groter dan één bevatten, is ook interessant. In dat geval zouden we een soort groepshuwelijk kunnen modelleren, waar John getrouwd zou kunnen zijn met drie vrouwen: Linda, Olivia en Mary, en waar Olivia getrouwd zou kunnen zijn met twee mannen, zoals dat is weergegeven in figuur 3. Alle karakteristieken van associaties houden ook in dat geval stand.




**FIGUUR 3. Waar beide associatie-einden een multipliciteit groter dan één bevatten, zouden we een groepshuwelijk kunnen modelleren.**

## SETS, ORDERED SETS, BAGS EN SEQUENCES

UML-associaties kunnen zelfs nog veel leuker zijn. UML biedt ook de mogelijkheid om aan te geven welk type verzameling gebruikt wordt voor een associatie-einde met een multipliciteit groter dan één. Je kunt één van de volgende vier types uitkiezen:

1. *Set*: ieder element mag slechts eenmaal in de verzameling vertegenwoordigd zijn. Dit is het default verzamelingstype voor een associatie-einde met een multipliciteit groter dan één.
2. *Ordered Set*: een set waarin de elementen zijn geordend. Er is een index-nummer voor ieder element. Merk op dat de elementen niet gesorteerd zijn, dus een element met een lager index-nummer is niet op wat voor manier dan ook groter of kleiner dan één met een hogere index. Een associatie-einde met dit type wordt in het diagram weergegeven door het toevoegen van de markering <ordered>.
3. *Bag*: een element mag meer dan één keer in de verzameling voorkomen. In het huwelijksvoorbeeld betekent dit bijvoorbeeld dat Mary twee keer met John getrouwd zou kunnen zijn. Wanneer één associatie-einde dit type heeft, dan moet het andere einde een bag of een sequence zijn. Het type wordt in het diagram weergegeven door <bag>.
4. *Sequence*: een bag waarin de elementen worden geordend. In het diagram wordt dat aangegeven met <sequence> of <seq>. Wanneer één einde van een

associatie dit type heeft, dan moet het andere einde een bag of een sequence zijn.

Wat zijn nu de effecten van het kiezen van verschillende verzamelingstypen voor een associatie-einde? We moeten er allereerst op wijzen dat de ABACUS-regels moeten standhouden, welk type je ook kiest. Het type van het associatie-einde is alleen relevant wanneer een element wordt toegevoegd aan de relatie, en dit is weer erg belangrijk voor de manier waarop associaties worden geïmplementeerd. In het diagram voor het groepshuwelijk hebben we het einde 'my wifes' gemarkeerd als een geordende set. Dit betekent dat de implementor van de *addWife* operatie in de class *Man* die een vrouw toevoegt aan de geordende set, moet beslissen hoe de set van echtgenoten geordend moet worden. De eenvoudigste optie is om 'new wife' als laatste toe te voegen aan de geordende set. De implementor van *addWife* moet er zeker van zijn dat de nieuwe echtgenote niet reeds gerepresenteerd is in de lijst met echtgenoten. Het OrderedSet type staat dit niet toe.

**ASSOCIATIES IMPLEMENTEREN** Het allerbelangrijkste bij het implementeren van associaties is om er zeker van te zijn dat de tweezijdige connectie altijd klopt. Deze moet altijd voldoen aan de ABACUS-regels. Dit betekent dat wanneer er iets verandert in de links, de implementatie zorg dient te dragen voor het opnieuw aanpassen van de referenties aan beide zijden van de associatie. Laten we beginnen met het de eenvoudigste case: de één op één associatie.

**EÉN OP ÉÉN ASSOCIATIE** Eén op één associaties kunnen geïmplementeerd worden met gebruik van twee velden (pointers) in de twee geassocieerde klassen. In klasse *Man* zou het type van het veld *Woman* moeten zijn, en vice versa. Omdat de associatie symmetrisch is, kan dezelfde implementatie gebruikt worden in beide klassen. Naast het veld hebben we get and set-operaties nodig: *getMyWife* en *setMyWife*. De body van de *getMyWife* operatie is eenvoudig; het retourneert de waarde van het veld, zoals hieronder wordt getoond. (*In dit geval gebruiken we de Java-syntax, maar de voorbeelden kunnen eenvoudig worden vertaald in een andere programmeertaal. De complete voorbeeldcode is beschikbaar op de website van dit blad, www.release.nl*).

```
public Woman getMyWife() {
   return f_myWife;
}
```

De body van de *setMyWife* operatie is complexer; het is verantwoordelijk om het nieuwe *Woman*-object ervan bewust te maken dat het op het punt staat te trouwen. Het zou erg handig zijn als we dat konden doen door gewoon de *setMyHusband*-operatie aan te roepen, die deel moet zijn van de *Woman*-klasse. Dit zou echter een oneindige loop veroorzaken, omdat *setMyHusband* op zijn beurt *setMyWife* opnieuw zou aanroepen. Daarom moeten we checken of het veld al op de juiste waarde is gezet. Dat gebeurt in de eerste regel van de body:

```
public void setMyWife(Woman element) {
   if ( this.f_myWife != element )
   {    // prevent infinite loop
      if ( this.f_myWife != null )
      {    // there is a previous wife!
         // remove the link with the
         previous wife
         this.f_myWife.z_internalRemoveFrom
         MyHusband( (Man)this );
      }
      this.f_myWife = element;
      // set the field to the new value
      if ( element != null ) {
         // make the new wife aware of the
         link
         element.setMyHusband( (Man)this );
      }
   }
}
```

Verder moeten we rekening houden met de mogelijkheid dat onze man al getrouwd is. Er zijn twee manieren om dat te implementeren. Of we checken of dit het geval is en zo ja, we weigeren iets te doen, of we zorgen dat de link met de vorige echtgenote wordt ontbonden. In de bovenstaande code hebben we gekozen voor de laatste optie. We gebruiken voor dit doel een speciale operatie: *z_internalRemoveFromMyHusband*. Deze operatie behoort alleen te worden aangeroepen vanaf *setMyWife*, daarom heeft het zo'n obscure naam. Het enige wat de operatie doet, is het *f_myHusband*-veld in het 'vorige echtgenote'-object naar null zetten.

```
public void z_internalRemoveFromMyHusband
(Man element) {
   this.f_myHusband = null;
}
```

Hiermee hebben we zeker gesteld dat alle betrokken elementen zich bewust zijn van de links die tussen hen bestaan. Als we bepaalde eisen aan de nieuwe echtgenote of echtgenoot moeten stellen (zo moet er overeenstemming zijn tussen de partners), is dit eenvoudig toe te voegen aan de set operaties. Eerst checken we wat er over de kandidaat-partner bekend is, en alleen wanneer het resultaat in orde is, kan de rest van de operatie uitgevoerd worden.

De meeste andere karakteristieken van de associatie worden ook door deze code geregeld. Zo wordt het *Use of Rolenames*-principe voor het partner-element geïmplementeerd in de get-operatie. Ten tweede bestaat de

link alleen als de velden van beide partners feitelijk aan elkaar refereren. Het enige wat niet wordt afgehandeld is de case waar één van de partners wordt verwijderd ('overlijden'). In dat geval moet de implementor van de *delete* operatie ervoor zorgen dat *setMyWife* of *setMyHusband* operatie van het partner-object wordt aangeroepen met *null* als parameter. Onze implementatie leunt volledig op garbage collection om verwijderde objecten af te handelen, waardoor er geen *delete* of *destroy*-operatie voorhanden is die deze afhandelt. De gegeven implementatie kan ook gebruikt worden wanneer één of beide associatie-einden optioneel zijn, dus wanneer de multipliciteit 0 of 1 is (0..1).

**Eén op veel associatie** We gaan nu onderzoeken hoe je een een op veel associatie moet implementeren. Hiervoor gebruiken we het voorbeeld van het polygame huwelijk, waar een vrouw met meerdere mannen kan trouwen. De implementatie van de *Man*-klasse verschilt niet zoveel van de één op één implementatie. Dat is logisch, want vanuit de *Man*-klasse is er niet zoveel veranderd. Hij is nog steeds met één vrouw getrouwd. De *Man*-klasse heeft een veld genaamd *f_myWife* van type *Woman*, en een *get and set*-operatie voor het veld. De body van de get-operatie is volledig gelijk aan de één op één case, en zelfs de body van de set-operatie komt in sterke mate overeen.

```
public void setMyWife(Woman element) {
   if ( this.f_myWife != element ) {
      if ( this.f_myWife != null ) {
         this.f_myWife.z_internalRemoveFrom
         MyHusbands( (Man)this );
      }
      this.f_myWife = element;
      if ( element != null ) {
         element.z_internalAddToMyHusbands
         ( (Man)this );
      }
   }
}
```

Het enige verschil is dat de nieuwe echtgenote het echtgenoot-object moet toevoegen aan de collectie in plaats van alleen de waarde vaststellen van het veld dat de echtgenoot vertegenwoordigt. Het veld dat de echtgenoten in de klasse *Woman* vertegenwoordigt, wordt *f_myHusbands* genoemd en heeft het type *java.util.Set*. Het wordt geïnitialiseerd naar een lege set. Dit is de implementatie van de speciale *add*-operatie in de klasse Woman:

```
public void z_internalAddToMyHusbands(Man
element) {
   this.f_myHusbands.add(element);
}
```

We gebruiken hier een speciale operatie die moet worden onderscheiden van de *addToMyHusbands*-operatie die eveneens beschikbaar is in de klasse *Woman*. Net als de *z_internalRemoveFromMyHusband*-operatie zou deze speciale operatie alleen moeten worden aangeroepen vanaf de *setMyWife*-operatie, zoals de laatste zorgdraagt voor de rest. In contrast daarmee mag de *addToMyHusbands*-operatie overal in het systeem gebruikt worden. In dat geval is het verantwoordelijk voor het juist opzetten van de links. De implementatie van de *addToMyHusbands* operatie luidt als volgt:

```
  public void addToMyHusbands(Man element) {
     if ( element == null ) {
        return;  // never add null to a
        collection
     }
     if ( this.f_myHusbands.contains(element)
) {
        return; // f_myHusbands is a set. If
                    the association end was a
                // bag or a sequence these
                   lines would not be present.
     }
     this.f_myHusbands.add(element);
     // the actual adding of the husband
     if ( element.getMyWife() != null ) {
          // a husband may have only one wife,
                       make sure that the link
             // of the old wife to this man is
                                        removed
        element.getMyWife().z_internalRemove
        FromMyHusbands(element);
     }
     // let the new husband know this object
     is its wife
     element.z_internalAddToMyWife
     ( (Woman)this );
  }
```

Laten we de code doornemen per statement. Allereerst, wanneer de operatie wordt aangeroepen met een null-waarde als parameter, moeten we niets doen. In de tweede plaats, wanneer de operatie wordt aangeroepen met een object dat al is opgenomen in de set echtgenoten, moeten we ook niets doen. Als we heel precies willen handelen, zouden we een waarschuwing moeten afgeven: "dit object is reeds aanwezig in de set echtgenoten", bijvoorbeeld door het afgeven van een exception. Wij kiezen voor de benadering dat je met dingen moet omgaan zoals met kinderen: streng, maar niet al te strict.

Het derde statement is de feitelijke toevoeging van de parameter aan de set 'echtgenoten'. Daarna moeten we zorgen voor de andere referenties. Als de nieuwe echtgenoot een eerdere echtgenote had, zou deze link verwijderd moeten worden. Opnieuw gebruiken we een speciale operatie genaamd *z_internalRemoveFromMyHus-*

*bands* om dit te implementeren. Het laatste statement zet de link op vanaf de kant van de nieuwe echtgenoot, daarbij gebruik makend van de speciale operatie *z_internalAddToMyWife*. Eerlijk gezegd zou deze operatie eigenlijk *z_internalSetMyWife* moeten heten, maar toen we dit deel van de Octopus code-generator schreven waren we een beetje lui en hebben we de naam van de operatie veranderd in de veel op veel case.

In de implementatie van de *Woman*-class zal een enkele set-operatie voor het veld *f_myHusbands* niet voldoende zijn. We moeten de volgende cases implementeren:

- een *set* operatie met een verzameling als parameter die de collectie van echtgenoten vaststelt als de gegeven verzameling.
- een *add*-operatie met een enkelvoudig object als parameter, zoals hiervoor is toegelicht, die één echtgenoot toevoegt.
- een *add*-operatie met een verzameling als parameter die alle elementen in de parameter verzameling toevoegt aan de set echtgenoten.
- een *remove*-operatie met een enkelvoudig object als parameter die een enkele echtgenoot verwijdert.
- een *remove*-operatie met een verzameling als parameter die alle elementen in de parameter-verzameling verwijderd uit de set echtgenoten.
- een *clear*-operatie die alle geassocieerde objecten verwijderd, waardoor de set echtgenoten leeg achterblijft.

Je kunt de implementatie van al deze operaties vinden in de sourcecode.

**VEEL OP VEEL ASSOCIATIE** De implementatie van de veel op veel case is natuurlijk identiek aan de implementatie van de 'veel'-kant in de één op veel situatie. Trouwens, de klassen aan beide einden kunnen vanwege de symmetrie op dezelfde manier worden geïmplementeerd als in de één op één situatie. In beide klassen hebben we het veld nodig, de *get*-operatie, en de *set*, *add* en *remove*-operaties voor elementen uit de collectie. Laten we eens kijken naar de implementatie van de *addToMyHusbands* operatie in de veel op veel case:

```
public void addToMyHusbands(Man element) {
   if ( element == null ) {
      return;
   }
   if ( this.f_myHusbands.contains(element) ) {
      return;
   }
   this.f_myHusbands.add(element);
   element.z_internalAddToMyWifes(
(Woman)this );
}
```

Je ziet dat het bijna identiek is aan de *addToMyHusbands*-operatie in de één op veel situatie. Wat ontbreekt is het deel dat zorgt voor het verwijderen van de link van de nieuwe echtgenoot met alle voorgaande echtgenotes. Vanwege de veel op veel semantiek mag deze link op z'n plek blijven.

Een andere interessante operatie, die volledig identiek is aan de één op veel case, is de *set*-operatie die de verzameling als een parameter ziet. Deze kan duidelijk verdeeld worden in drie delen. Allereerst moeten we de links met alle huidige echtgenoten verbreken. Daarna zetten we het veld naar de nieuwe waarde. Als derde zetten we de link op met de nieuwe echtgenoten.

> Er zijn maar een beperkt aantal verschillende configuraties voor een associatie

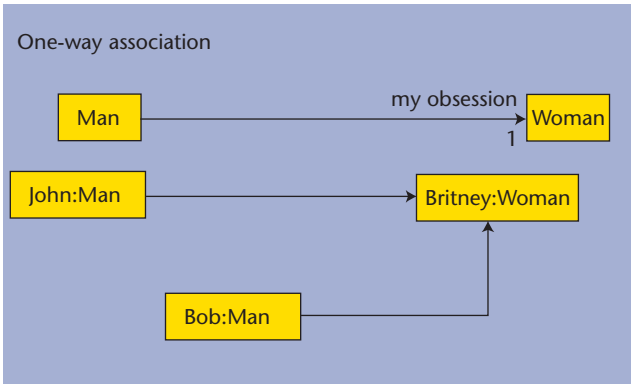
```
public void setMyHusbands(Set elements) {
   if ( this.f_myHusbands != elements ) {
      // remove the link with all current
         husbands
      Iterator it = this.f_myHusbands.
      iterator();
      while ( it.hasNext() ) {
         Man x = (Man) it.next();
         x.z_internalRemoveFromMyWifes(
         (Woman)this );
      }
      this.f_myHusbands = elements;
      if ( f_myHusbands != null ) {
         // set up the link with the new
            husbands
         it = f_myHusbands.iterator();
         while ( it.hasNext() ) {
            Man x = (Man) it.next();
            x.z_internalAddToMyWifes(
            (Woman)this );
         }
      }
   }
}
```

Tenslotte, omdat we de links tussen de echtgenoten en echtgenotes consistent willen houden, hebben we de *get*-operaties geïmplementeerd om een niet-aanpasbare lijst te retourneren. Als je een echtgenoot of echtgenote wilt toevoegen of verwijderen, moet je de operaties gebruiken die voor dat doel werden gemaakt. Dit dwingt de ABACUS-regels af, geassocieerde objecten zijn zich bewust van het toevoegen of verwijderen van links.

```
public Set getMyWifes() {
     if ( f_myWifes != null ) {
        return Collections.unmodifiableSet(f_
        myWifes);
     } else {
        return null;
     }
  }
```

**Eenzijdig navigeerbare assocaties** Nu we de complexe aspecten van associaties hebben behandeld, gaan we terug naar een eenvoudiger case: degene waar slechts één van de associatie-einden is gemarkeerd als navigeerbaar. In het diagram wordt dit getoond door een pijlpunt bij het associatie-einde. Als er een pijlpunt aanwezig is, betekent dit dat het andere einde niet-navigeerbaar is. Dat betekent dat het object dat een instantie is van de klasse waar de pijl naartoe wijst, zich niet bewust is van de relatie. Eigenlijk is dit gewoon een manier om een reguliere pointer te modelleren.

## Traditionele UML modelleer-tools genereren geen correcte implementatie van associaties




**Figuur 4. Voorbeeld van een eenzijdig navigeerbare associatie.**

Gebruik deze configuratie dus niet wanneer je een relatie als een huwelijk wilt modelleren, gebruik het alleen wanneer je een gerichte relatie wilt modelleren, zoals tussen een popster en zijn of haar fans. Ik denk namelijk niet dat iemand als Britney Spears iedere fan persoonlijk zou willen kennen. Het is in UML trouwens ook mogelijk om beide eindpunten niet-navigeerbaar te maken, maar dat zou geen enkele functie hebben.

Het implementeren van een eenzijdig navigeerbare associatie betekent dat de klasse waar de pijl naar wijst, in figuur 4 is dat *Woman*, niet wordt aangetast. De andere klasse, *Man*, heeft alleen een veld nodig en eenvoudige *get* en *set* operaties, of wanneer de multipliciteit groter dan één is, de operaties die het toevoegen en verwijderen van geassocieerde objecten implementeren.

Onder modelleerders is een debat gaande of de rolnaam en multipliciteit al dan niet moet worden toegevoegd aan de kant die niet-navigeerbaar is. Vanaf het gezichtspunt van de implementatie van de associatie is deze informatie irrelevant. Het wordt nergens in de code gerepresenteerd. Desalniettemin zou degene die het model leest enig inzicht kunnen verwerven als je ze wel zou toevoegen. In het voorbeeld zouden we de rolnaam *fan* kunnen toevoegen, en de multipliciteit nul tot veel (0..*), en je zou dan weten dat er mogelijk meerdere instanties van de klasse *Man* zijn die een pointer bevatten naar een enkele instantie van de klasse *Woman*.

**Generatie van code voor associaties** Uit het bovenstaande mag duidelijk zijn dat het implementeren van associaties geen triviale zaak is. Is het niet verbazingwekkend hoeveel code je moet implementeren voor zo'n simpel diagram, of hoeveel code er verandert wanneer je besluit om een '1' in het diagram te veranderen in een '1..*'? Dit laat zeker de kracht van modelleren zien.

Er zijn echter maar een beperkt aantal verschillende configuraties voor een associatie. In dit artikel hebben we vier implementaties laten zien: de één op één, één op veel, veel op veel, en de eenzijdig navigeerbare configuratie. Andere configuraties komen van het combineren van deze met een associatie-klasse. Verder kan iedere associatie met dezelfde configuratie worden geïmplementeerd op dezelfde manier. Daarom pleiten wij voor het genereren van code, in het bijzonder voor het implementeren van associaties.

Wanneer je associaties handmatig moet implementeren, moet je dezelfde handelingen steeds opnieuw doen, wat niet erg interessant is. Bovendien maak je sneller fouten wanneer je het handmatig doet (en verveeld raakt). Wanneer de modelleerder besluit dat de multipliciteiten veranderd moeten worden, kun je weer helemaal opnieuw beginnen. Dat is precies de reden waarom het implementeren van associaties een goed voorbeeld is van de kracht van de Model Driven Architecture (MDA). MDA-tools die code genereren vanuit een UML-model, genereren ook de implementatie van de associaties. Voorbeelden van zulke tools zijn Octopus en Eclipse Modeling Framework (zie kader 'MDA-tools').

**Tools** In dit artikel hebben we de output van Octopus behandeld. Er zijn verschillende implementaties mogelijk, zolang die overeenstemmen met de ABACUS-regels. Zo hanteert EMF een totaal andere

## MDA-tools

*Octopus*

De Octopus-tool kan gedownload worden van http://www.klasse.nl/octopus/index.html. Naast het genereren van alle code die nodig is om de associaties te implementeren, kan dit tool een user interface prototype genereren en een XML-storage faciliteit vanaf een UML-model. Dit maakt het mogelijk om de applicatie te draaien en instanties van de klassen in het model te creëren.

*EMF*

Het Eclipse Modeling Framework is een modeling framework en code-generatie faciliteit voor het bouwen van tools en andere applicaties die gebaseerd zijn op een gestructureerd datamodel. Vanuit een modelspecificatie beschreven in XMI, levert EMF-tools en runtime-support voor het produceren van onder meer een set van Java-klassen voor het model. Meer informatie vindt u op http://www.eclipse.org/emf/.

benadering dan Octopus. Achter de schermen is er een compleet notificatie framework dat het partner object op de hoogte stelt van het aanmaken van de link. Wij vinden dit framework nodeloos gecompliceerd, en we hebben aangetoond dat een eenvoudiger implementatie ook correct draait. We onderschrijven het principe dat de code die je genereert bijna hetzelfde moet zijn als de code die je handmatig zou schrijven. De gegenereerde code moet in ieder geval op zijn minst leesbaar zijn en gemakkelijk te begrijpen.

Merk op dat veel van de traditionele UML modelleer-tools ook claimen dat ze code genereren, maar ze genereren niet een correcte implementatie van associaties. Tools als Together, Poseidon of Rational Rose genereren alleen een paar pointers en laten de rest aan jou over. Het zou goed zijn als de klanten druk zouden uitoefenen op de aanbieders om hun tools correcte implementaties van associaties te laten genereren. Dat kan eenvoudig worden gerealiseerd en het zal de benodigde inspanningen om een UML-model te genereren verminderen.

### Mind teasers

We sluiten dit artikel af met een aantal *mind teasers* die kunnen helpen bij het soepel leren gebruiken van associaties in een model. Ga er gewoon eens mee spelen en probeer te begrijpen wat een bepaald model betekent voor de instanties van de gegeven klassen.

1. Onderzoek hoe je een huwelijk kunt modelleren met één klasse: *Person*, zonder de klassen *Man* en *Woman* te gebruiken. Probeer ook een homosexueel huwelijk te modelleren.
2. Een andere interessante situatie is wanneer er twee associaties zijn tussen dezelfde twee klassen. Probeer een situatie te modelleren waar een persoon voor een bank werkt, en zich daarmee dus in een werkgever-werknemer relatie bevindt, terwijl die persoon tegelijkertijd een client is van de bank. Wat nu als die persoon werkt voor verschillende banken?

> Elk paar objecten dat verbonden is via een associatie heeft een aantal karakteristieken, samengevat in de ABACUS-regel

Tenslotte willen we er nog op wijzen dat het introduceren van de klassen *Man* en *Woman* om mensen te vertegenwoordigen niet altijd een goed idee is. We gebruikten ze in dit artikel om de UML-associatie uit te leggen, maar in het echte leven liggen de zaken doorgaans gecompliceerder, wat een andere classificatie zou kunnen vereisen.

*Jos Warmer is partner bij Ordina.*
*Anneke Kleppe is onderzoeker aan de Universiteit Twente.*