Van alle onderdelen van een UML-model wordt vaak alleen het klassendiagram gebruikt als input voor codegeneratie. Soms wordt code gegenereerd vanuit toestandsdiagrammen en/of activiteitendiagrammen. De mogelijkheden tot code generatie zijn hierbij afhankelijk van de kwaliteit van het model en de gebruikte tools. Alle andere zaken dient de ontwikkelaar met de hand aan de gegenereerde code toe te voegen.

thema

Query's in UML

Haal meer code uit een UML-model

Dit artikel beschrijft de mogelijkheden van OCL, de query-taal uit de UML-standaard voor codegeneratie. Met behulp van OCL kunnen alle benodigde query's op modelniveau gedefinieerd worden. Door code-generatie vanuit OCL is het niet meer nodig om arbeidsintensieve query's met de hand op codeniveau te maken. Dit artikel is geen complete inleiding tot OCL. We introduceren de taal door middel van een aantal voorbeelden, waarmee een goed begrip verkregen kan worden van de mogelijkheden van OCL. Een volledige beschrijving van OCL kunt u vinden in "The Object Constraint Language: Getting Your Models Ready for MDA"[1].

We laten naast OCL-query's ook zien hoe de equivalente Java-code eruit ziet. Alle in dit artikel getoonde Java-code is vanuit OCL gegenereerd met behulp van Octopus. Octopus is een open source tool, die op basis van Eclipse met behulp van een aantal plug-ins een complete OCL-ontwikkelomgeving vormt. Octopus is gratis te downloaden op http://www.klasse.nl/english/research/octopus-intro.html [4].

OCL: de UML-querytaal Naast de visuele overzichtsdiagrammen kent UML ook een complete query-taal. Deze taal heet OCL, wat staat voor Object Constraint Language. Deze naam dekt de lading dus niet. OCL is niet slechts een constraint-taal, maar een complete query-taal met minstens dezelfde uitdrukkingskracht als SQL. Knappe wetenschappers hebben dit netjes bewezen, dus dat zit wel goed.

Zoals bekend is UML een standaard van de OMG (Object Management Group). Buiten OMG-kringen is OCL lange tijd een van de best bewaarde geheimen van UML geweest. Weinig mensen kenden de taal en de ondersteuning van tools was mondjesmaat. Binnen de OMG heeft het gebruik van OCL inmiddels een grote vlucht genomen. Nagenoeg alle standaarden van de OMG bevatten tegenwoordig een flinke dosis OCL, hetgeen de eenduidigheid en kwaliteit van de standaarden sterk ten goede komt. Ook de nieuwe QVT (Query-View-Transformations) standaard van de OMG, waarin een taal gedefinieerd wordt waarmee MDA (Model Driven Architecture)-transformaties beschreven kunnen worden leunt stevig op OCL.

De laatste jaren neemt de bekendheid van OCL toe, en is ook een groeiend aantal tools beschikbaar dat het gebruik ervan ondersteunt. Leidend op dit gebied is Borland, die OCL volledig ondersteunt, maar ook bedrijven als IBM/Rational, Interactive Objects, Compuware, etc. bieden tegenwoordig OCL ondersteuning.

Hoe wordt OCL gebruikt? Om een lang verhaal kort te maken: overal in UML waar je een expressie tegenkomt kan OCL gebruikt worden. Voorbeelden hiervan zijn er te over:

- Invarianten in een klassenmodel
- De body van query-operaties
- Pre- en postcondities bij operaties of use cases
- Keuzes in activiteitendiagrammen
- Keuzes in sequence diagrammen
- Guards in toestandsdiagrammen
- Toestandsinvarianten in toestandsdiagrammen
- Afleidingsregels voor afgeleidde attributen en associaties
- Initiële waarden voor attributen en associatie-einden
- Etc. etc.

In dit artikel leggen we de focus op het gebruik van OCL voor het specificeren van query's over klassenmodellen. Alle bovenstaande soorten van gebruik zijn als query's te beschouwen. Zo is een invariant of een preconditie niets anders dan een query die een Boolean-

resultaat oplevert. De body van een query-operatie is simpelweg een query die het resultaat van de operatie definieert.

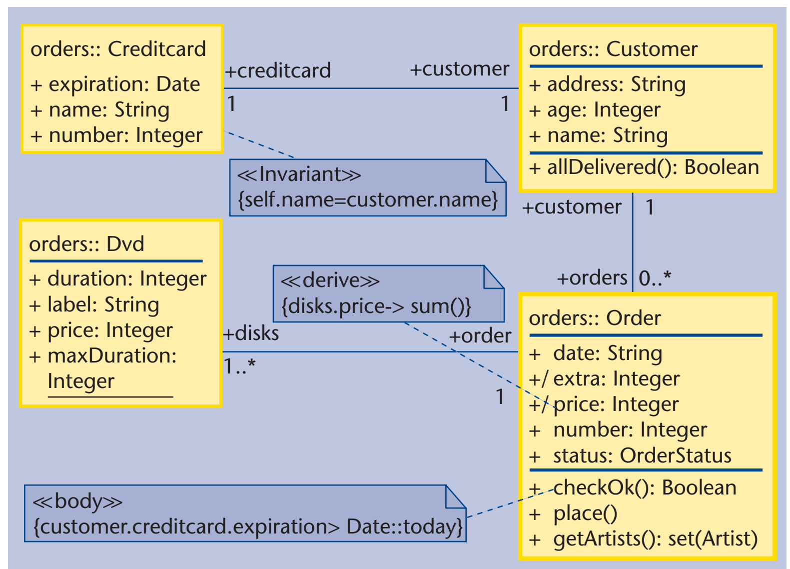
In een UML-klassenmodel worden OCL-query's getoond in een zogenaamde notebox. De query is met een stippellijn verbonden met zijn context, het element waar de query betrekking op heeft. Het stereotype binnen de notebox geeft aan om wat voor type query het gaat. Zo is de bovenste query in het model een « invariant » voor de klasse *Creditcard*, hetgeen aangeeft dat de expressie altijd waar moet zijn voor alle instanties van deze klasse. De middelste query beschrijft de afleidingsregel voor het attribuut *price*, en de onderste query bevat de body van de operatie *checkOk()*.

Een visueel UML-diagram is met name handig om goed overzicht te houden. Als zo een diagram vol komt te staan met query's, komt dat de leesbaarheid bepaald niet ten goede. In de praktijk blijkt dat het handig is om de OCL-query's naast het visuele model tekstueel te schrijven. Om deze reden definieert de OCL-standaard naast de visuele syntax ook een OCL bestandsformaat. De bovenstaande query's zien er in het OCL-bestandsformaat als volgt uit.

```
package Orders

   context Creditcard
   inv: self.name = customer.name

   context Order::checkOk() : Boolean
   body: customer.creditcard.expiration >
Date::today

   context Order::price
   derive: disks.price->sum()

endpackage
```

De stippellijnen zijn vervangen door het keyword *context* met daarachter de volledige naam van het element waar de query bij hoort. Verder wordt de query zelf voorafgegaan door een keyword dat aangeeft om wat voor type query het gaat. De ontwikkelaar krijgt op deze wijze het beste van twee werelden. Overzicht door middel van visuele UML-modellen, en precisie met behulp van tekstuele OCL-bestanden.

OCL TOEGEPAST Alle voorbeelden in dit artikel zijn query's die gekoppeld zijn aan klassenmodellen. We beschrijven hier daarom eerst het klassenmodel dat we vervolgens bij alle voorbeelden zullen gebruiken. Het voorbeeld betreft een eenvoudige DVD-shopapplicatie. Er zijn *Customers*, die elk precies één *Creditcard* hebben. *Customers* kunnen een willekeurig aantal *Orders* hebben. Een *Order* bestaat vervolgens uit één of meer *DVD's* met daarop één of meer *Clips*. *Clips* zijn



FIGUUR 1. OCL query's in een klassendiagram.

gesorteerd in een *Catalog* en worden door een willekeurig aantal *Artists* uitgevoerd. Verder worden er enkele enumeraties gedefinieerd welke als type voor attributen gebruikt worden.

In het voorbeeld in figuur 2 staat de operatie *getCheapClips()* van de klasse *Catalog*. Deze operatie selecteert uit alle *clips* behorende bij een *Catalog* de clips die goedkoper zijn dan de parameter *maxPrice*. In OCL wordt dat als volgt opgeschreven:

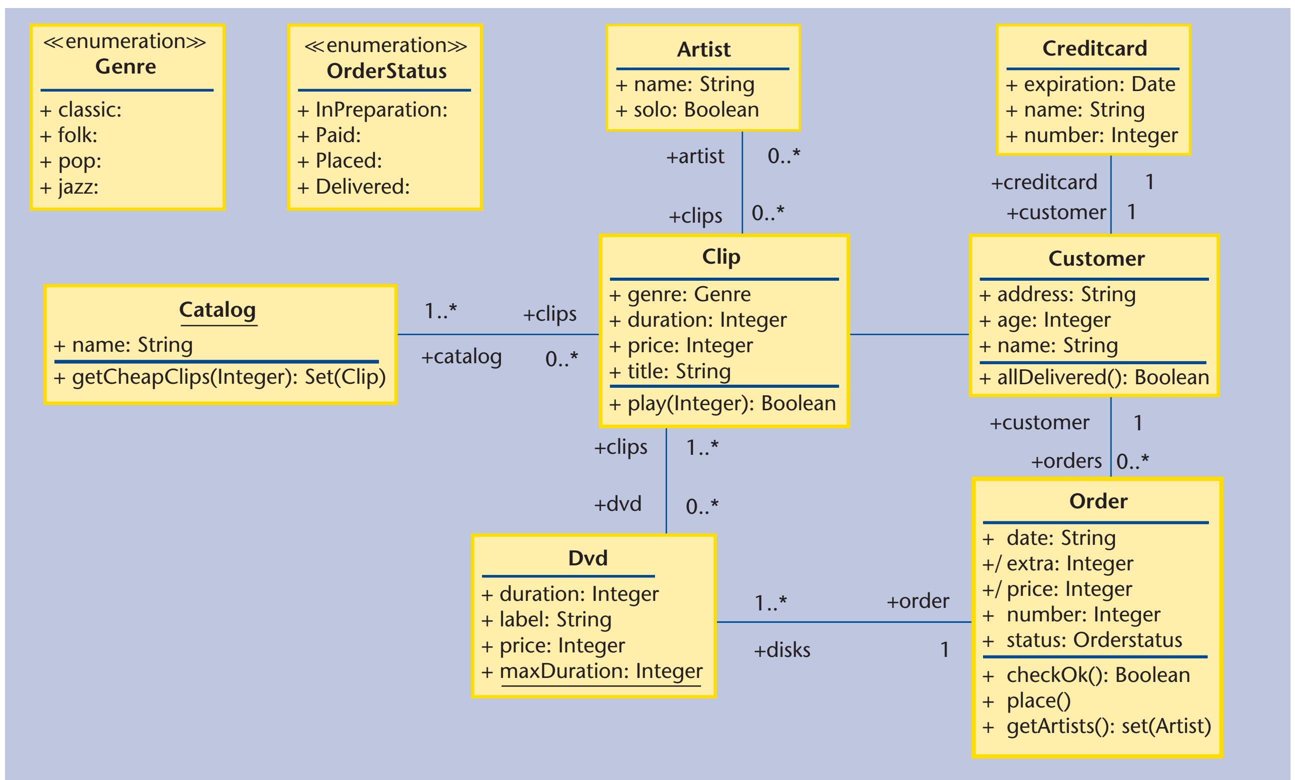
```
context Catalog::getCheapClips(
         Integer maxPrice) : Set(Clip)
body: clips->select( c | c.price < maxPrice)
```

Hierbij is *clips* een navigatie over de associatie van *Catalogus* naar *Clip*, waarbij de rolnaam aan de zijde van *Clip* gebruikt wordt. Deze navigatie levert de bij de *Catalog* behorende verzameling van *Clips* op. Op deze wijze kan in een OCL-query genavigeerd worden over alle associaties in het model. De standaard OCL-operatie *select()* selecteert vervolgens uit de verzameling clips degene waarvan de *price* kleiner dan de meegegeven *maxPrice* is. Dit levert de gevraagde verzameling van clips op.

De OCL-notatie is gebaseerd op objectgeoriënteerde principes. Mensen met een Smalltalk-achtergrond zullen bovenstaande notatie dan ook direct herkennen. Omdat smaken verschillen is het ook mogelijk om OCL-expressies met een andere notatie te schrijven. Zo hebben we in [1] een alternatieve notatie gedefinieerd welke voor mensen met een SQL-achtergrond eenvoudiger te begrijpen is. In deze notatie wordt de bovenstaande query als volgt geschreven:

Advertentie Microsoft

Advertentie Microsoft



Figuur 2. KLASSENMODEL VAN EEN EENVOUDIGE DVD-SHOPAPPLICATIE.

```
context Catalog::getCheapClips(
         Integer maxPrice) : Set(Clip)
body: SELECT c : Clip FROM clips
      WHERE c.price < maxPrice
```

Deze notatie is weliswaar geen onderdeel van de UML-standaard, maar de UML-standaard geeft wel de mogelijkheid om zelf een eigen notatie te definiëren. In het vervolg gebruiken we de standaardnotatie, maar het is goed om te weten dat alternatieven mogelijk zijn.

Van OCL naar Java In het kader van de MDA-aanpak willen we zoveel mogelijk van onze code via transformaties uit modellen te genereren. We gaan hier verder niet in op de code die gegenereerd kan worden uit het UML-klassenmodel, maar leggen de nadruk op de toegevoegde waarde die OCL op dit gebied kan bieden.

OCL biedt de mogelijkheid om query's op modelniveau te definiëren, en ze vervolgens volledig naar code te transformeren. In dit artikel geven we voorbeelden van codegeneratie naar Java, maar generatie naar andere talen als C# of SQL is natuurlijk ook mogelijk, zie bijvoorbeeld[2, 3]. Alle getoonde Java-code is daadwerkelijk gegenereerd met Octopus[4].

Body van query-operaties Query-operaties in UML zijn operaties die geen neveneffecten hebben. OCL-expressies hebben ook geen neveneffecten en zijn dus uitermate geschikt om de body van een query-operatie mee te specificeren. In het model van de DVD shop kunnen we bijvoorbeeld de operaties *checkOk()* en *getArtists()* volledig in OCL specificeren.

Eenvoudige query's Om te beginnen specificeren we de operatie *checkOk()*, welke relatief eenvoudig is. Deze operatie checkt of de creditcard van de customer die de order geplaatst heeft niet reeds verlopen is.

```
context Order::checkOk() : Boolean
body: self.customer.creditcard.expiration.
isAfter(utilities::Date::today())
```

De Java code voor de operatie `checkOk()` kan vervolgens geheel gegenereerd worden.

```
/** Implements the user defined operation
 * '+ checkOk() : Boolean '
 */
public boolean checkOk() {
  return this.getCustomer().getCreditcard().
        getExpiration().isAfter(Date.today());
}
```

Zoals te zien wijkt de Java-code niet veel af van de OCL-query. Wel is zijn er verschillen die interessant zijn om nader te bekijken. In OCL worden de attributen en associaties zoals ze in UML gedefinieerd zijn direct gebruikt. In Java-code is het een goede gewoonte om

attributen privé te maken en vervolgens een publieke *getter*- en een *setter*-methode te gebruiken om de waarde te manipuleren. In de gegenereerde Java-code zijn alle referenties naar de UML-attributen en associaties netjes getransformeerd naar de equivalente Java-expressies, gebruik makende van de Java *getter*-methodes.

Complexe query's over verzamelingen Een voorbeeld van een complexere query is de definitie van de operatie *getArtists()* van klasse *Order*. Om alle artiesten die bij een order horen te vinden moeten we verschillende verzamelingen aflopen. Eerst vinden we vanuit de order de verzameling *disks*, dan kunnen we voor elke disk in die verzameling de *clips* vinden die op die disk horen. Dan hebben we beschikking over de verzameling van alle clips bij alle disks in die order. Om nu de gevraagde set artiesten te vinden moeten van elk element uit deze laatste verzameling de bijbehorende artiesten 'opvragen'. Dit klinkt heel ingewikkeld, maar in OCL kun je het simpel opschrijven:

```
context Order::getArtists() :
                Set(catalog::Artist)
body: disks.clips.artist->asSet()
```

Jammer genoeg merk je weer dat dit een complexe query is als je de equivalente Java-code moet schrijven. In de Java-code moet je namelijk een while-loop met een iterator gebruiken om elk element van een verzameling te bereiken. Voor bovenstaande query betekent dat je twee verschillende while-loops nodig hebt, zoals te zien in het volgende codefragment.

```
/** Implements the user defined operation '+
getArtists() : Set(Artist) '
  */
public Set getArtists() {
    return Stdlib.collectionAsSet
                          (collect4Artists());
}

/** Implements ->collect( i_Clip : Clip |
 * i_Clip.artist )
 */
private List collect4Artists() {
    List /*(Artist)*/ result = new ArrayList(
/*Artist*/);
    Iterator it = collect3Clips().iterator();
    while ( it.hasNext() ) {
        Clip i_Clip = (Clip) it.next();
        Object bodyExpResult =
                           i_Clip.getArtist();
        result.addAll(
                (Collection)bodyExpResult );
    }
    return result;
}
```

```
/** Implements ->collect( i_Dvd : Dvd |
 * i_Dvd.clips )
 */
private List collect3Clips() {
    List /*(Clip)*/ result =
                    new ArrayList( /*Clip*/);
    Iterator it = this.getDisks().iterator();
    while ( it.hasNext() ) {
        Dvd i_Dvd = (Dvd) it.next();
        Object bodyExpResult = i_Dvd.
getClips();
        result.addAll(
                    (Collection)bodyExpResult );
    }
    return result;
}
```

In de Java-code hebben we ervoor gekozen om voor iedere iterator met bijbehorende while-loop een aparte methode te definiëren. Een alternatief zou zijn om alle while-loops genest te schrijven, dat levert naar ons idee echter code op die lastiger te lezen is, zeker wanneer de nesting meerdere niveaus diep is.

Initiële waardes Het volgende voorbeeld toont het gebruik van OCL-query's voor het definiëren van de initiële waarde van een attribuut.

```
context DVD::label
init: 'default title'
```

Het lijkt vrij simpel om dit om te zetten naar Java-code en in dit geval klopt dat ook, want de initiële waarde is een heel simpele expressie. Maar ook complexere expressies kunnen als initiële waarde worden gebruikt. Bovendien moet het zetten van de initiële waarde in elke constructor van de klasse-DVD op dezelfde manier geregeld worden. Hieronder laten we één constructor als voorbeeld zien.

```
/** Default constructor for Dvd
 */
public Dvd() {
    this.setLabel( ("default title") );
    if ( usesAllInstances ) {
        allInstances.add(this);
    }
}
```

Derivation rules In UML kunnen attributen, maar ook associaties als *afgeleid* aangegeven worden. Dit geeft aan dat de waarde van zo een eigenschap niet op zichzelf staat, maar altijd afgeleid kan worden uit andere attributen en associaties. Op zich nuttig om dit aan

te geven, maar de werkelijk toegevoegde waarde zit hem in de afleidingsregel zelf. Hoe wordt het attribuut of associatie-einde afgeleid?

Hieronder volgen twee afleidingsregels. De eerste geeft aan dat de prijs van een order bepaald wordt door de prijs van de bestelde disks plus een toeslag. De tweede geeft aan dat de toeslag wordt bepaald door de grootte van de order. Als de totale prijs van de bestelde disks hoger is dan 15 euro dan is er geen toeslag, anders is de toeslag 2 euro.

```
context Order::price
derive: disks.price->sum() + extra

context Order::extra
derive: if disks.price->sum() < 15 then 2
                                      else 0
         endif
```

Hier hebben we wederom te maken met het werken met verzamelingen. Hoewel de OCL-query's eenvoudig lijken, is ook hier veel Java code nodig om ze te implementeren.

```
/**  Implements the getter for feature '+
extra : Integer'
  */
public int getExtra() {
   return ((sum2() < 15) ? 2 : 0);
}

/** Implements .sum() on disks->collect(
 * i_Dvd : Dvd | i_Dvd.price )
 */
private int sum2() {
    int result = 0;
    Iterator it = collect1().iterator();
    while ( it.hasNext() ) {
        Integer elem = (Integer) it.next();
        result = result + elem.intValue();
    }
    return result;
}

/** Implements ->collect( i_Dvd : Dvd |
 * i_Dvd.price )
 */
private List collect1() {
  List /*(Integer)*/ result =
                new ArrayList( /*Integer*/);
  Iterator it = this.getDisks().iterator();
  while ( it.hasNext() ) {
                Dvd i_Dvd = (Dvd) it.next();
     Object bodyExpResult =
              new Integer(i_Dvd.getPrice());
     if ( bodyExpResult != null )
                result.add( bodyExpResult );
  }
  return result;
}
```

INVARIANTEN OCL-query's die gebruikt worden als invariant, horen in principe altijd waar te zijn voor alle instanties van de klasse waarvoor de invariant gedefinieerd is. In principe – want het is niet altijd duidelijk wanneer een invariant gecheckt moet worden. In heel kritische systemen zullen alle invarianten gecheckt moeten worden wanneer er ook maar iets in het object wijzigt, dus bijvoorbeeld na elke operatie-aanroep. In minder kritische systemen is dit gewoonweg teveel overhead en kan worden volstaan met het checken op bepaalde tijden. Ook kan ervoor gekozen worden om invarianten alleen tijdens het testen te gebruiken en JUnit-tests te genereren waarin de invarianten gecontroleerd worden.

```
context Clip
inv: duration > 0
```

Bovenstaande is een invariant, die aangeeft dat de lengte van een *Clip* altijd groter dan nul moet zijn. De expressie *duration > 0* dient altijd *true* op te leveren voor alle objecten van type *Clip*. In Octopus levert deze query de volgende Java-code op.

```
  /** Implements self.duration > 0 */
public void invariant_Clip2() throws
InvariantException {
    boolean result = false;
    try {
        result = (this.getDuration() > 0);
    } catch (Exception e) {
        e.printStackTrace();
    }
    if ( ! result ) {
        String message =
             "invariant self.duration > 0 "
           + "is broken in object '"
           + this.getIdString()
           + "' of type '"
           + this.getClass().getName()
           + "'";
        throw new InvariantException(this,
                                    message);
    }
}
```

De Java-code voor het uitrekenen van de OCL-query is hier simpel: *this.getDuration() > 0*. We hebben hiervoor al complexere code gezien. De code rond deze expressie is hier echter veel interessanter. In deze code vallen de volgende zaken op:

- Een invariant in OCL wordt een methode in Java. Deze keuze is gemaakt zodat de ontwikkelaar zelf kan kiezen waar en wanneer de invariant gecontroleerd gaat worden. Ook is de code, behorende bij een invariant, duidelijk herkenbaar.
- De methode wekt een exception als de invariant gebroken is.
- De exception bevat naast het object waarvoor de invariant gebroken is ook extra informatie in de vorm van een string.

De drie bovenstaande punten zijn allen bewuste ontwerpkeuzes bij de codegeneratie. Een invariant hoeft niet per definitie als methode geïmplementeerd te worden. Ook zou een invariant methode in plaats van een exception een returnwaarde kunnen hebben welke aangeeft of de invariant gebroken is. In alle gevallen is het in ieder geval zo dat codegeneratie zorgt voor een volledig eenduidige wijze van codering.

CONCLUSIE Het gebruik van OCL-query's in UML is een verrijking van de visuele UML-modellen. Het zorgt voor een grotere precisie en compleetheid van de resulterende UML-modellen. Belangrijker nog, OCL-query's zijn volledig executeerbaar, dat wil zeggen dat er altijd honderd procent code uit OCL-query's gegenereerd kan worden. Dit betekent dat de moeite die in het opstellen van OCL-query's gestoken wordt zichzelf altijd terugbetaald. Het hoge abstractieniveau van de OCL-query's ten opzichte van programmeertalen zoals Java betekent bovendien dat de productiviteit hiermee omhoog gaat: één regel OCL komt vaak overeen met tientallen regels Java.

De Java-code in dit artikel is gebaseerd op een aantal ontwerpkeuzes. Andere keuzes zijn mogelijk, maar ook dan blijft honderd procent codegeneratie mogelijk. Door het genereren van code uit OCL is tevens gegarandeerd dat de ontwerpkeuze overal consistent toegepast wordt, hetgeen de kwaliteit en consistentie ten goede komt.

Tool ondersteuning voor OCL is zowel als open source, alsook vanuit commerciële bedrijven steeds meer voorhanden. De hier genoemde tools zijn slechts enkele voorbeelden waarbij we vooral hebben gekeken naar de variatie in talen waarnaar generatie plaatsvindt (Java, C#, Delphi, SQL). Een korte zoektocht op Internet levert al snel een grote lijst aan interessante tools op. Hierbij is het wel zaak om goed te kijken welke tools het best aansluiten bij de wensen en mogelijkheden van de omgeving waarin ze gebruikt gaan worden.

Referenties

1 Anneke Kleppe, Jos Warmer, *The Object Constraint Language Second Edition, Getting Your Models Ready for MDA*, 2003, Addison-Wesley
2 Dresden OCL Toolkit op http://dresden-ocl.sourceforge.net/
3 C# OCL compiler op http://www.ewebsimplex.net/csocl/
4 Octopus website op http://www.klasse.nl/english/research/octopus-intro.html

Jos Warmer is partner bij Ordina.
Anneke Kleppe is eigenaar van Klasse Objecten en werkt als onderzoeker aan de Universiteit Twente.