

Metamodellierung

Markus Völter (voelter@acm.org)

völter – ingenieurbüro für softwaretechnologie

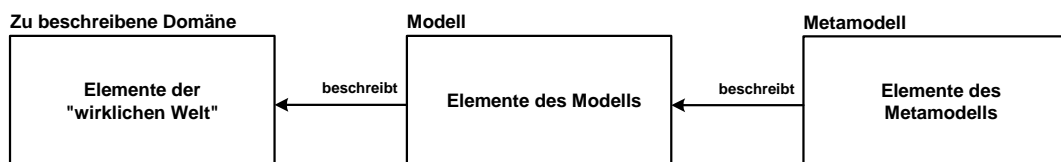
Ziegeläcker 11, 89520 Heidenheim

www.voelter.de

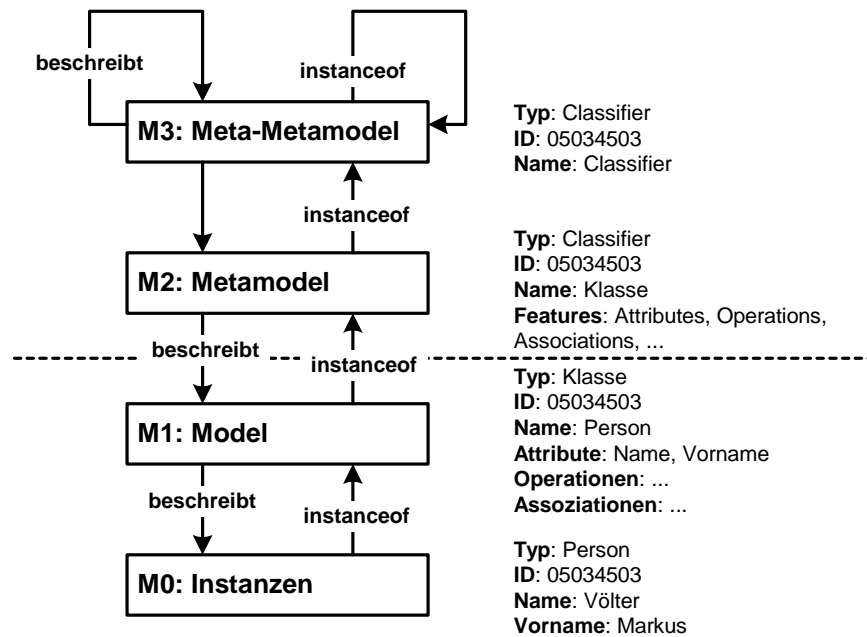
Metamodellierung ist die wohl wichtigste Voraussetzung für modellgetriebene Entwicklung abgesehen vielleicht von Codegenerierung (sonst entsteht ja nichts ausführbares). Metamodellierung ist Basis für Domänenspezifische oder architekturzentrierte Modellierung, Toolanpassung, Modellvalidierung, Modelltransformationen und Codegenerierung. Anhand dieser Liste sollte klar sein, warum wir diesem Thema einen eigenen Abschnitt im Buch widmen.

Was ist Metamodellierung

Metamodellierung besteht aus den beiden Worten *Modellierung* und der griechischen Vorsilbe *meta*. Meta bedeutet dabei soviel wie *über*, *nach*, *neben* oder *zwischen*. Metamodelle sind Modelle von Modellen. Anders ausgedrückt: Modellierung beschreibt eine bestimmte Domäne mittels eines Modells. Dieses Modell wird mittels einer bestimmten Sprache, der Modellierungssprache beschrieben. Diese Sprache enthält eine Menge von Sprachkonstrukten mit definierter Syntax und Semantik. Metamodellierung dient nun dazu eine solche Modellierungssprache mittels eines Modells zu beschreiben. Dazu wird (konsequenterweise) eine Metamodellierungssprache verwendet, die ihrerseits wieder aus Modellelementen mit definierter Syntax und Semantik besteht. Um den Kreis also zu schließen: Metamodellierung beschreibt (Meta)Modelle, deren Domäne Modellierungssprachen sind.



Die OMG hat zu diesem Zwecke vier Metaschichten definiert, die im Folgenden beschrieben werden.



Metamodellierung, UML und die MOF

Im Rahmen der Softwareentwicklung wird man in aller Regel nicht damit beginnen, eine komplett neue M2-Sprache auf Basis der MOF zu definieren. Vielmehr wird man typischerweise mit dem UML-Metamodell beginnen und dieses passend erweitern.

Um diese Erweiterung durchzuführen gibt es drei Möglichkeiten:

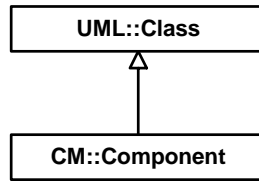
- Erweiterung auf Basis des formalen Metamodells der UML
- Erweiterung mittels Stereotypen (mit UML 1.x-Mitteln)
- Erweiterung mittels Stereotypen (mit UML 2.x-Mitteln)

Wir werden jede dieser Varianten im Folgenden betrachten. In der Praxis, wenn man wird man – schon der besseren Toolunterstützung wegen – den Profilmechanismus verwenden. Meines Erachtens stellt der metamodellbasierte Ansatz aber den konzeptionell saubereren dar.

Erweiterung auf Basis des Metamodells

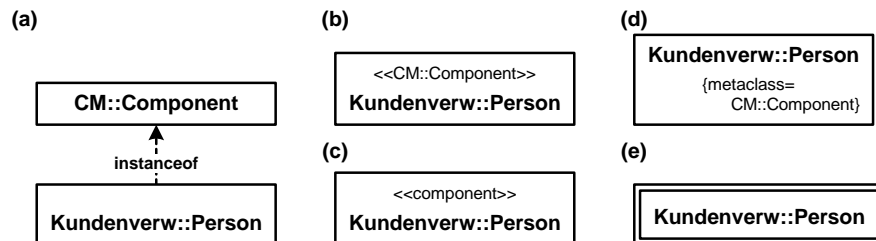
Bei dieser Art der Erweiterung wird das Metamodell der UML erweitert. Dabei verwenden wir (wie immer bei der Modellierung) die Sprachmittel des nächsthöheren Metalevels, hier also M3, MOF. Eine solche Erweiterung kann also nur im Rahmen von Tools erfolgen, die ein explizites, auf der MOF basierendes Metamodell besitzen.

Um also eine „eigene Art von Klasse“ zu definieren erstellt man eine neue MOF-Klasse die von der UML-Metaklasse `UML::Class` erbt. Dies ist im Folgenden dargestellt.



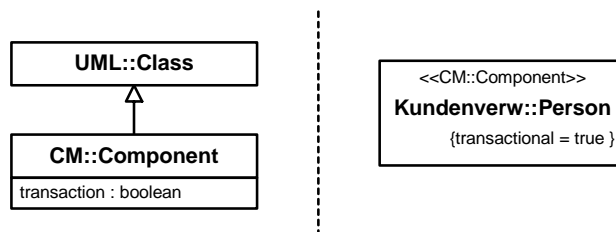
Hier wird also ein neues Sprachkonstrukt definiert, die *CM::Component*. Diese ist eine Unterklasse des Elementes *Class* aus der UML. Zu beachten ist, daß hierbei der Vererbungsmechanismus der MOF zum Einsatz kommt, nicht der der UML!

Noch eine Anmerkung zur Verwendung von Tools: Theoretisch kann man jedem selbstdefinierten Sprachelement (hier also *MyExtension::Class*) eine eigene grafische Repräsentation zuweisen (siehe (e) unten). Dies ist oft allerdings nicht pragmatisch möglich. Es bieten sich daher andere Darstellungsmöglichkeiten an, die meisten basieren auf Stereotypen.



(a) zeigt die Klasse *Kundenverwaltung::Person* direkt als Instanz der Metaklasse *MyExtension::Class*. (b) verwendet den Namen der Metaklasse als Stereotyp; (c) eine per Konvention verabredete Kurzform, (d) einen Tagged Value und (e) zu guter Letzt eine eigene grafische Notation.

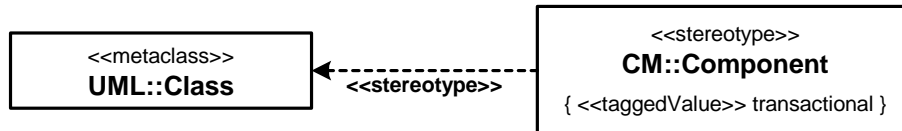
Die *CM::Component* unterscheidet sich zunächst – abgesehen vom formalen Typ – nicht von einer UML-Klasse, da sie ja keine Attribute und Operation hinzufügt bzw. überschreibt. Man kann aber einer eigenen Metaklasse auch neue Attribute geben. Dies wird dann in den verwendenden Modellen üblicherweise mittels eines Tagged Value dargestellt.



Man beachte, daß diese Art der Metamodellerweiterung nicht nur mit UML Modellen funktioniert, sondern mit allen auf der MOF basierenden Modellierungssprachen.

Erweiterung mittels Stereotypen in UML 1.x

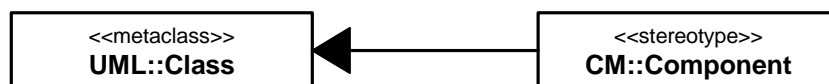
Die Erweiterung mittels Stereotypen ist eine Funktionalität die spezifisch die UML zur Verfügung stellt. Der Mechanismus hat formal nichts mit der MOF zu tun, man verläßt nie den M2-Level. Insofern funktioniert dieser Mechanismus der Erweiterung auch nur mit der UML, andere MOF-basierte Sprachen müßten ihre eigenen Erweiterungsmechanismen definieren. Das folgende Diagramm zeigt die Definition des Stereotypen *CM::Component* mit dem *transactional* Tagged Value.



Hierbei sind einige Dinge zu bemerken. Zum einen ist der Tagged Value nicht typisiert. In UML 1.x sind diese automatisch Strings. Zum zweiten braucht man sich keine Gedanken machen, wie man diese Notation in UML darstellt, weil sie ja ein natives UML Konzept darstellt (ein Stereotyp wird logischerweise als Stereotyp dargestellt). Zu guter letzt ist zu sagen, daß das Mittel um einen Stereotyp zu definieren selbst wieder Stereotypisierung ist (und nicht irgendein Metamodell!). Der Stereotyp wird notiert indem das Klassensymbol mit dem Stereotyp *stereotype* stereotypisiert wird.

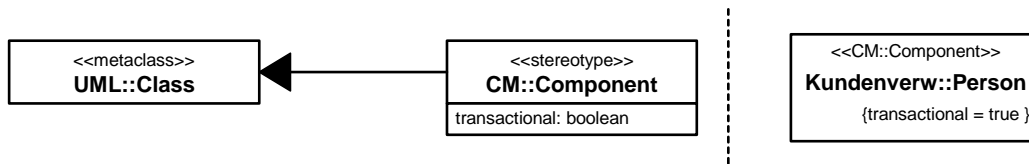
Erweiterung mittels UML 2.x

In UML 2.x wurden die Sprachmittel mittels derer Stereotypisierung erfolgt erweitert bzw. umgestellt. Dies ist geschehen im Rahmen von Profilen, die in der nächsten Sektion beschrieben werden. Zentral dabei ist das Konzept der *Extensions*. Eine Extension ist dabei ein neues Symbol (und damit ein neues Sprachkonstrukt) im Rahmen der UML. Es wird dargestellt durch den „ausgefüllten Vererbungs Pfeil“ (er ist hier bewußt etwas zu groß dargestellt um Verwechslungen mit dem Assoziationspfeil zu vermeiden).



Es sei nochmals darauf hingewiesen, daß es sich hierbei nicht um Vererbung, Implementierung oder eine stereotypisierte Abhängigkeit oder Assoziation handelt sondern um ein komplett neues UML Sprachmittel. Dieses ist im UML Metamodell auch sauber formal definiert (siehe Profiles-Kapitel).

Ein Stereotyp kann Attribute haben; diese werden dann im Modell in dem der Stereotyp verwendet wird als Tagged Values dargestellt.



Ein weiterer Unterschied zu UML 1.x ist der daß ein Modellelement nun mehrere Stereotypen gleichzeitig haben kann. Es hat dann die Attribute aller Stereotypen als Tagged Values¹.

UML-Profile

Profile dienen zur Anpassung bzw. Erweiterung der UML an fachliche oder technische Domänen. Man könnte auch sagen, UML ist keine Sprache sondern eine Sprachfamilie. UML-Profile sind Elemente - also konkrete Sprachen - dieser Familie. Eine Zielvorstellung ist die, dass UML-Tools und Generatoren Profile wie PlugIns verarbeiten können, d.h. man ‚lädt‘ zunächst ein spezielles fachliches oder technisches Profil und dazu gehörige Transformationen. Anschließend kann man auf Basis des Profils modellieren und dann z.B. Quellcode generieren. Damit dies in der Praxis gut funktioniert, benötigt man offensichtlich eine klare Trennung von Modell, Profil, Transformationen und Werkzeug. Zu diesem Zweck definiert die OMG einen Profilmechanismus für die UML.

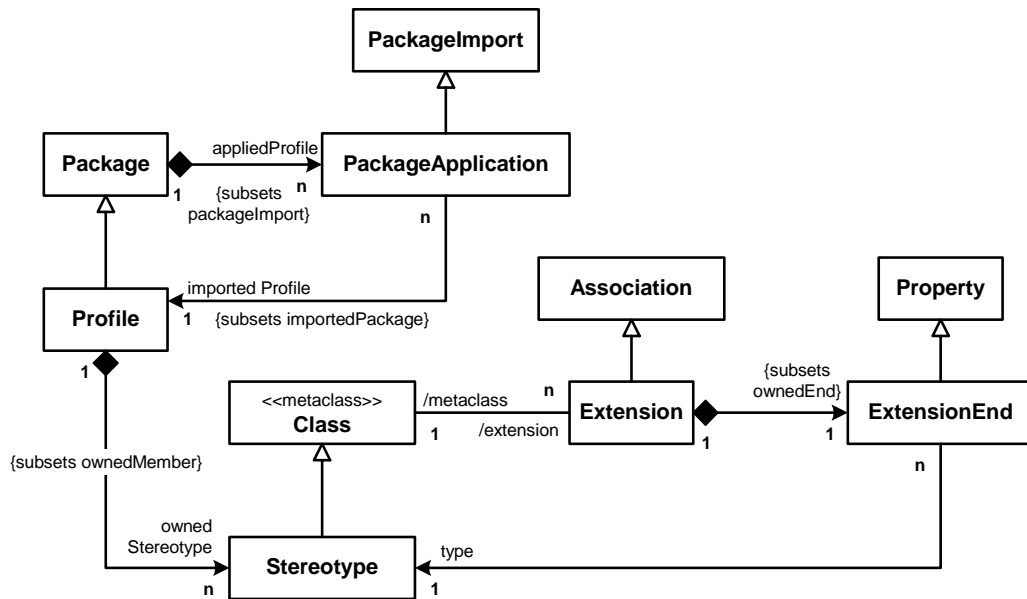
Auch hierbei ist wieder zu erwähnen, daß es sich um einen auf UML-Metamodellebene definierten Mechanismus handelt der formal nichts mit der MOF und Meta-Metamodellierung zu tun hat. UML Profile bestehen prinzipiell aus drei Dingen: Stereotypen, Tagged Values und Constraints². Dabei können Profile die bereits von der UML definierten Constraints nur erweitern (also weiter einschränken) nicht aber lockern. In UML 1.x waren Profile im Prinzip nur locker verbal definiert. In UML 2.0 ist der Profilmechanismus basierend auf dem UML Metamodell formal definiert. Dabei findet sich auch eine Definition des Extension-Konzeptes welches im vorherigen Abschnitt erwähnt wurde.

Im Folgenden das Metamodell der Profildefinition aus der UML 2.0 Spezifikation (welches nebenbei auch als Beispiel für Metamodellierung dienen kann). Namespaceangaben fehlen, weil alles Teil des UML Metamodells ist³.

¹ Noch eine Formalität: Genau genommen sind die Tagged Values keine Tagged Values mehr, sondern die Darstellung der Attribute der Stereotyps. Sie werden aber – weil sie genauso aussehen – weiterhin als Tagged Values bezeichnet.

² UML 2.0 definiert das formal etwas anders – siehe unten.

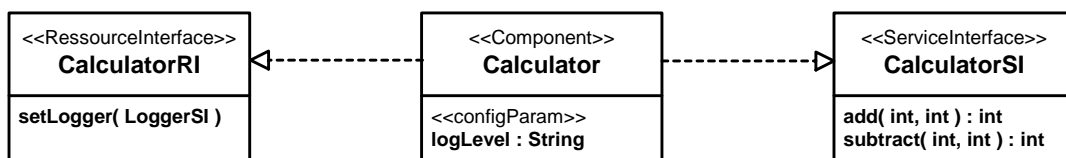
³ Weiterhin fehlen der Einfachheit halber auch einige Constraints.



Anwendungsbeispiel

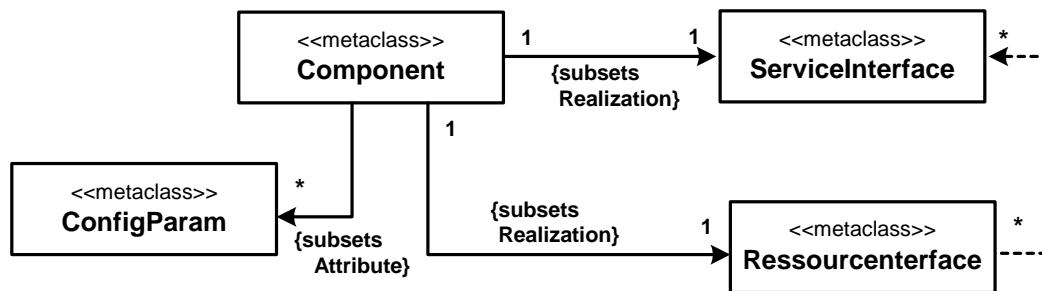
Als Anwendungsbeispiel soll eine vereinfachte Komponenteninfrastruktur dienen [VSW02], und zwar eine, die für Kleingeräte und Embedded-Systeme anwendbar ist [MV02]. Zentraler Bestandteil von Anwendungen die auf dieser Infrastruktur beruhen sind – naheliegenderweise – Komponenten. Es macht also Sinn im Rahmen der Architekturdefinition anzugeben, was eine Komponente ist. Wir beginnen also mit der Definition eines Metamodells für Komponenten dieser Infrastruktur.

Zunächst wollen wir zum Ausdruck bringen, daß eine Komponente ein Serviceinterface implementiert welches definiert, welche Operationen die Komponente „als Dienstleistung“ für andere Komponenten anbietet. Des weiteren soll die Komponente ein sogenanntes Ressourceninterface besitzen. Dieses definiert, welche Ressourcen (also andere Komponenten, Datenbankverbindungen, Thread Pools) die Komponente benötigt um selbst lauffähig zu sein. Das Ressourceninterface bietet eine Setter-Operation für jede Ressource an, wobei per Definition immer das Serviceinterface der Ressource als Typ verwendet wird. Zu guter letzt soll eine Komponente noch eine Reihe sogenannter Konfigurationsparameter haben. Dies sind – der Einfachheit halber – einfach Attribute der Komponentenklasse die vom Typ *String* sein müssen, da sie beim Starten des Systems aus einer Konfigurations-Datei geladen werden sollen. Hier zunächst ein Beispiel:



Es wird eine einfache Rechnerkomponente definiert, deren Serviceinterface eine Operation *add* und eine Operation *subtract* hat; des weiteren erwartet sie im Ressourceninterface eine Referenz auf eine *Logger* Komponente, und der Rechner hat einen Konfigurationsparameter, den *loglevel*.

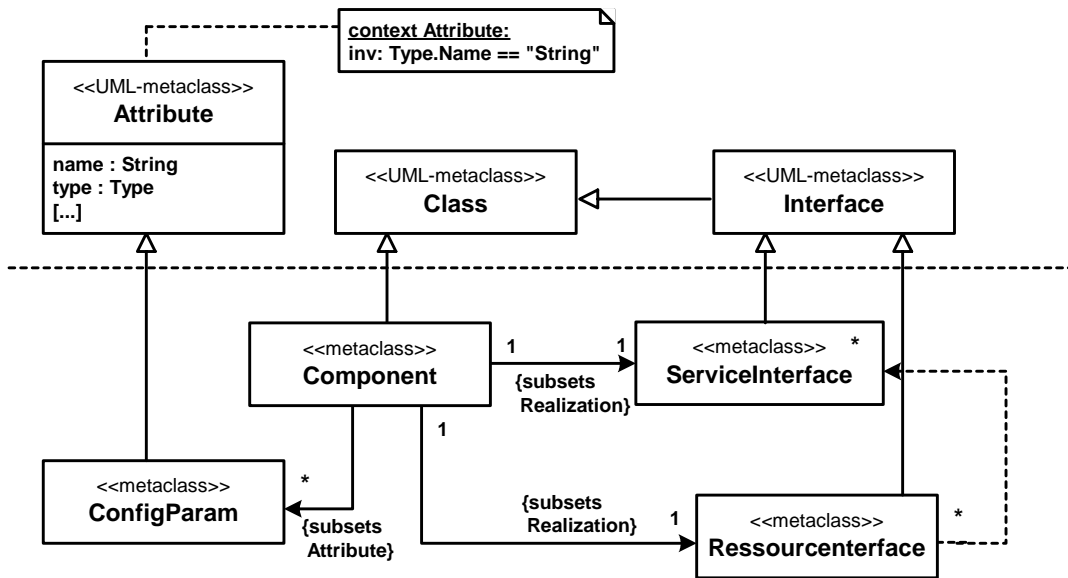
Das folgende UML Diagramm zeigt nun das Metamodell dieser Architektur.



Es bringt zum formell Ausdruck, was wir oben verbal beschrieben haben – zumindest zum Teil. Die Ankopplung eines Instanzdiagrammes (*Calculator*) an das Metamodell passiert mittels Stereotypen: Wenn wir zum Ausdruck bringen wollen, daß z.B. *logLevel* eine Instanz des Metatyps *ConfigParam* ist, so geben wir diesem einfach den entsprechenden Stereotyp. Gleiches gilt bspw. für *Calculator*, das den Stereotyp *Component* trägt.

Verfeinerung des Metamodells

Ziel unserer Unternehmung hier ist es, aus Modellen (wie dem *Calculator*) Architektur- (also Metamodell-) konformen Code zu generieren. Zu diesem Zweck muß der Generator und Modellvalidierer das Metamodell kennen. Um nicht ein komplett eigenes Metamodell entwickeln zu müssen verwenden wir das UML Metamodell als Basis. Wir erinnern uns an die Aussage „Ein Konfigurationsparameter ist einfach ein Attribut von Typ *String*“. Etwas anders ausgedrückt: Es gibt im UML Metamodell die (Meta-)Klasse *Attribute*. Diese hat ein Attribut namens *Type*. Wir sagen hier nichts anderes, als daß die (Meta-)Klasse *ConfigParam* eine Unterklasse von *Attribute* ist, wobei gilt, dass deren Attribut *Type* den Wert *String* haben muß. Das unten folgende UML Diagramm bringt dies zum Ausdruck. Man beachte dabei die Verwendung von OCL zur Definition der Constraint, daß der *Type* eines *ConfigParams* immer *String* sein muß. Wir können nun sinngemäß für die anderen Elemente unseres Metamodells vorgehen. Die nächste Abbildung zeigt das Ergebnis.



Dieses Metamodell ist nun an das UML Metamodell „angekoppelt“. Dies ist wichtig für die Integration in den Codegenerator, weil dieser standardmässig mit dem UML Metamodell arbeitet – wir brauchen daher also kein komplett neues Metamodell bauen sondern nur das bestehende anpassen. Wie funktioniert nun diese Anpassung? Nun, im Prinzip genauso, wie im Diagramm oben.

Implementierung des Metamodells für den Generator

Das Metamodell des Generators (wir verwenden hier das b+m Generator Framework [BMWeb]) ist in Java implementiert. Es gibt also z.B. eine Klasse `de.bmiag.genfw.meta.Class` (die Packageangabe sparen wir uns zukünftig!) die immer dann vom Generator instanziiert wird, wenn in einem vom Generator verarbeiteten Modell eine UML-Klasse vorkommt. Dito für Attribute: für jedes im Modell vorkommende Attribut instantiiert der Generator eine Objekt der Klasse `Attribute`. Damit sollte klar sein, wie eine Metamodellanpassung aussieht: Wir bauen eine Unterklasse der entsprechenden Metaklasse und sagen dem Generator, wenn eine Instanz dieser neuen Metaklasse im Modell vorkommt, möge er doch bitte diese neue Metaklasse instantiiieren. Hier das Beispiel für den `ConfigParam`:

```
package metamodel;
public class ConfigParam extends Attribute {
}
```

Über eine (hier nicht gezeigte) Konfigurationsdatei sagen wir dem Generator, daß alle UML-Attribute die den Stereotyp `ConfigParam` haben in Wirklichkeit Konfigurationsparameter sind, er also deshalb die Unterklasse `ConfigParam` statt `Attribute` instantiiieren soll. Aus Sicht des Generators ist dies kein Problem, denn wie immer in der OO Programmierung kann ja auch eine Instanz einer Unterklasse verwendet werden, wenn eine Variable mit der Oberklasse getypt ist (Polymorphismus).

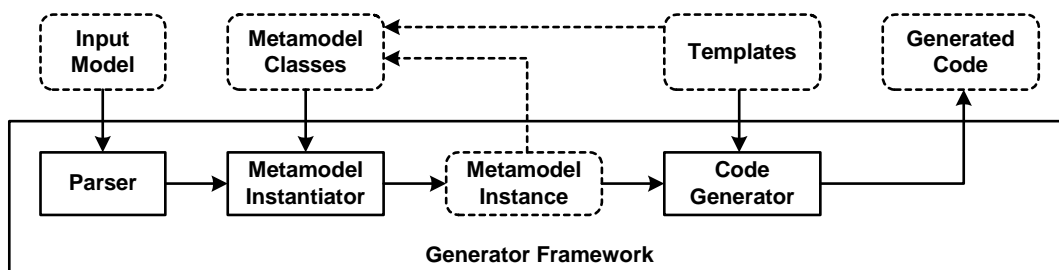
Nun bietet uns diese Klasse *ConfigParam* noch nicht viel Nutzwert. Insbesondere fehlt noch die Constraint, daß der Typ von *ConfigParams* immer *String* sein muß. Um derartige Constraints zu überprüfen besitzen alle Klassen des Metamodells eine Operation *CheckConstraints* die vom Generator aufgerufen wird, wenn das gesamte Metamodell instantiiert ist. Stellt diese Operation ein Problem fest, so wirft sie eine *DesignException* die nachfolgend dem Entwickler gemeldet wird – das verarbeitete Modell ist nicht konform zum verwendeten Metamodell. Hier der Code für *CheckConstraints()* der Klasse *ConfigParam*:

```
public String CheckConstraints() throws DesignException {
    if ( !Type.Name.toString().equals("String") ) {
        throw new DesignException( "ConfigParam Type not String" );
    }
    return super.CheckConstraints();
}
```

Auf die gleiche Art und Weise erstellen wir nun Metaklassen für *Component*, *ServiceInterface* und *ResourceInterface*. Die *CheckConstraints*-Implementierung mag dort etwas komplexer werden, aber das Prinzip ist dasselbe.

Funktionsprinzip des Generators

Um zu verstehen was bis jetzt passiert ist und wie die im folgenden erklärten *Templates* funktionieren, sollten wir zunächst einen Blick auf das Funktionsprinzip des Generators werfen. Das folgende Bild zeigt die grobe Struktur:



Als Eingabedaten verwendet der Generator ein Modell. Dieses wird vom Parser geparst; der daraus entstehende Parsebaum wird dann vom Metamodellinstantiator unter Verwendung des konfigurierten Metamodells instantiiert. Das Eingabe (Modell-) Format ist dabei austauschbar, da verschiedene Parser im Generator verwendet werden können. Hier (wie wohl in den meisten Fällen) kommt XMI zum Einsatz, welches unter anderem zur Beschreibung von UML Modellen verwendet werden kann. Nach der Instantiierung des Metamodells kann basierend auf der Metamodellinstanz die eigentliche Codegenerierung stattfinden. Dazu werden Templates verwendet.

Templates zur Codegenerierung

Eine Template besteht grundsätzlich aus zwei Aspekten: Dem Code, der über die Metamodellinstanz iteriert sowie dem zu generierenden Quellcode ggfs. mit Zugriff auf Attribute des Metamodells. Das folgende Stückchen Template-Code definiert eine

Template namens *Root* die auf den Metamodelltyp *Component* paßt. Die Template gibt an, daß für jede Komponente eine Datei mit dem Namen der Komponente angelegt werden soll, die dann später mit Code gefüllt wird.

```
«DEFINE Root FOR Component»
  «FILE Name".java"»
  «ENDFILE»
«ENDDDEFINE»
```

Interessant ist der Ausdruck *Name* innerhalb der *File* Deklaration: hier wird das Attribut *Name* der Instanz der Komponente ausgelesen, auf der wir gerade arbeiten. Wir greifen also auf das Modell – die Metamodellinstanz – zu. Dazu muß die Metaklasse *Component* ein öffentliches Attribut namens *Name*, oder eine öffentliche Methode *String Name()* besitzen. Im nächsten Beispiel legen wir nun innerhalb dieser Datei eine Java Klasse an, wieder mit dem Name der Komponente die wir grade bearbeiten:

```
«DEFINE Root FOR Component»
  «FILE Name".java"»
  public class «Name» {
  }
  «ENDFILE»
«ENDDDEFINE»
```

Der zu generierende Code steht dabei einfach außerhalb der französischen Anführungszeichen. Diese dienen auch als Escape-Character um innerhalb des zu generierenden Codes auf das Metamodell – hier der Name der Komponente – zuzugreifen. Um nun das Beispiel der Konfigurationsparameter wieder aufzunehmen, sei hier noch gezeigt, wie dieses realisiert wird. Generell wollen wir ja für alle Attribute (nicht nur *ConfigParams*) der Komponente ein entsprechendes Attribut in der Klasse anlegen:

```
[...]
  public class «Name» {
    «EXPAND ParamDefinition FOREACH Attribute»
  }
[...]
```

```
«DEFINE ParamDefinition FOR Attribute»
  «Type» «Name»;
«ENDDDEFINE»
```

Dies geschieht hier dadurch, daß für jedes Attribut eine Template namens *ParamDefinition* aufgerufen wird – diese ist dann direkt darunter definiert. Es wird also für jedes Attribut der Komponente im Modell ein Attribut im Code angelegt. Interessant wird es für *ConfigParams*. Diese sind ja auch Attribute, sie werden also ganz normal wie andere Attribute mit behandelt, es wird ein Attribut im Code angelegt. Allerdings wollen wir für diese *ConfigParams* auch je noch eine Operation *configure<ParamName>* anlegen. Dazu definieren wir die folgende Template:

```
«DEFINE ParamDefinition FOR ConfigParam»
  «Type» «Name»; // CconfigParam
  public void configure«CapitalizedName»( «Type» val ) {
    «Name» = val;
  }
}
```

«ENDEDEFINE»

Hier sind zwei Dinge zu beachten: Zum einen definieren wir hier eine Template mit dem gleichen Name wie vorher (*ParamDefinition*), aber für einen Untertyp von *Attribute*, nämlich *ConfigParam*. Das Generatorframework verwendet also bei der Templateexpansion automatisch die für den betreffenden Typ spezialisierteste Template! Wir haben damit Polymorphismus auf Template-Ebene erreicht.

Ein letztes Beispiel des Zugriffs aus Templates auf das Metamodell: Wir sprechen hier das Attribut *CapitalizedName* der Metaklasse *ConfigParam* an. Dies ist standardmäßig in der Metaklasse *Attribute* nicht vorhanden, wir definieren es also auf unserer *ConfigParam* Metaklasse:

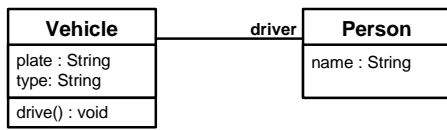
```
package metamodel;
public class ConfigParam extends Attribute {
[...]
    public String CapitalizedName() throws DesignException {
        return Name().toString().substring(0,1).toUpperCase()+
            Name().toString().substring(1);
    }
}
```

Modelltransformationen

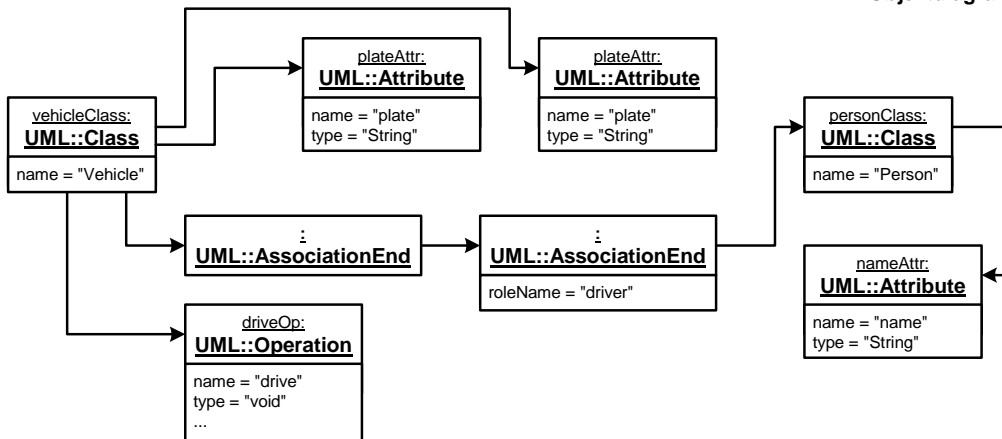
Modelltransformationen überführen ein Modell in ein anderes. Im Rahmen der MDA ist dies neben der Codegenerierung die wohl wichtigste Technologie. In der derzeitigen Praxis spielt Modelltransformation in Ermangelung von Sprachen und Tools praktisch keine Rolle. Im Bereich der Forschung wird allerdings intensiv daran gearbeitet. Die OMG hat dazu einen Request for Proposal ausgeschrieben (QVT RfP).

Grafische Notationen sind gerade erst im Entstehen, wie auch für die textuellen gibt es noch keinen Standard. Die uns bekannten Proposals haben aber alle gemeinsam, daß sie so weit wie möglich auf vorhandenen UML Konzepten und Notationen aufbauen. Unten ein Beispiel welches UMLX verwendet (ein nicht-offizielles QVT Proposal von Ed Willink)..

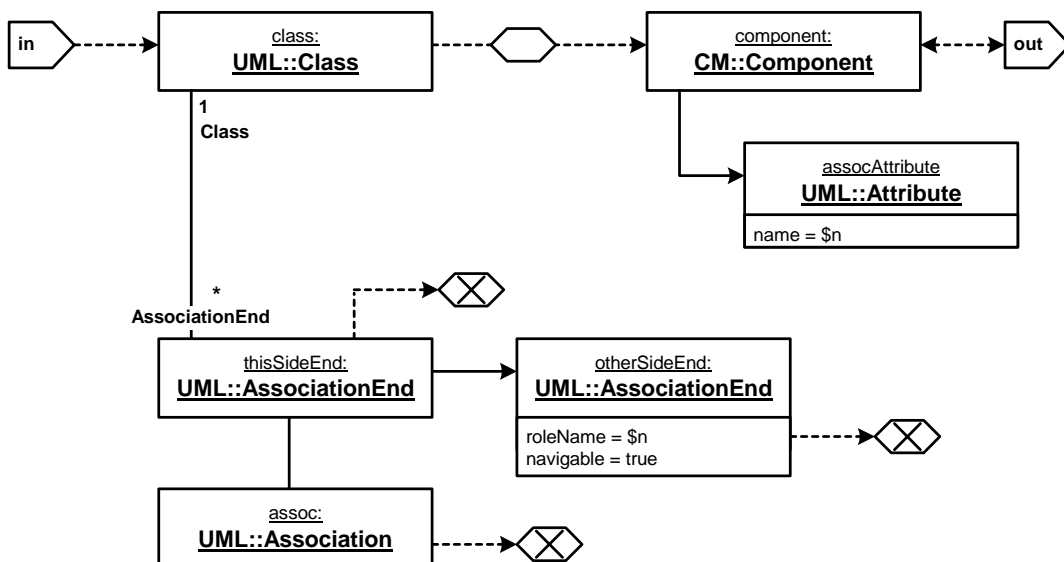
Zunächst aber ein kleiner Exkurs zum Thema Klassendiagramme und Objekt- (oder Instanz-) Diagramme in UML. Aufgrund der bisherigen Diskussionen zum Thema Metamodellierung sollte klar sein, daß sich jedes UML Klassendiagramm als Objektdiagramm des Metalevels darstellen läßt. Das folgende Beispiel sollte dies verdeutlichen.



Klassendiagramm
Objektdiagramm



Die Transformationsdefinitionen in UMLX sind nun mittels einer erweiterten Form von Objektdiagrammen erstellt. Objektdiagramme deshalb, weil eine Transformation ja zunächst eine Selektion der Modell-Objekte im zu transformierenden Klassendiagramm durchführen muß. Das folgende Diagramm zeigt eine beispielhafte Transformation – im Folgenden wird erläutert was diese bedeutet.



Als Eingangsdaten werden alle die Objekte gewertet, die von dem *in*-Symbol aus per Assoziationen erreichbar sind, also *class*, *thisSideEnd*, *otherSideEnd* und *assoc*. Nun wird die Klasse (mit dem Rollennamen *class*) in eine Komponente *component* transformiert, symbolisiert durch das Evolutionssymbol (die gestrichelte Linie mit dem Diamant). Dieses

Objekt ist gleichzeitig die Ausgabe der Transformation, symbolisiert durch die Verbindung zum *out*-Symbol. Weiterhin wird für jede Assoziation die die Eingangsklasse hat ein Attribut in der Zielkomponente mit demselben Namen angelegt (wir vernachlässigen hier mal den Typ). Die anderen Objekte (*thisSideEnd*, *otherSideEnd* und *assoc*) sind im Zielmodell nicht mehr vorhanden, symbolisiert durch das *Removal*-Symbol. Man beachte die Verwendung der freien Variable n die während der Selektion an den Rollennamen gebunden wird. Man beachte, daß dies nur für Assoziationsenden geschieht, die navigierbar sind. Alle in der Transformation nicht explizit erwähnten Modellelemente bleiben dabei „unberührt“, d.h. der Name der Komponente wird gleich sein wie der der Ausgangsklasse, sie wird auch alle Attribute haben die die Ausgangsklasse hatte.

Referenzen

- MV02 Markus Völter; *Small Components*;
<http://www.voelter.de/data/pub/SmallComponents.pdf>
- MV03a Markus Völter; *Program Generation – A survey of tools and techniques*;
<http://www.voelter.de/data/presentations/ProgramGeneration.zip>
- MC03b Markus Völter; *Patterns for Program Generation*;
<http://www.voelter.de/data/pub/ProgramGeneration.pdf>
- VS03 Völter, Stahl; *Architektur und Generierung*; iX 03/2003
- BMWeb b+m AG, *B+M Generator Framework*; <http://www.architectureware.de>
- VSW02 Völter, Schmid, Wolff; *Server Component Patterns*; Wiley & Sons, 2002