Metamodellbasierte Codegenerierung in Java

Markus Völter, voelter@acm.org, www.voelter.de

Codegenerierung

Codegenerierung bezeichnet die automatische Erstellung von Quelltext aus üblicherweise abstraktereren, oft domänenspezifischen Modellen. Üblicherweise sollen damit Ziele erreicht werden wie Vermeidung des Schreibens von immer gleichem Code, Architekturkonformität, Performanzsteigerung, oder eben die Möglichkeit, Dinge auf Ebene eines Modelles spezifizieren zu können und sich um die Details der Implementierung nicht kümmern zu müssen. Dieser Artikel geht nun nicht auf das für und wider von Codegenerierung, Integration von generiertem und nicht-generiertem Code ein. Er enthält auch keinen Überblick über die verschiedenen technischen Möglichkeiten wie Code generiert werden kann. Dazu gibt es andere Quellen [MV03a], [MV03b], [VS03]. Vielmehr soll dieser Artikel eine sehr effiziente Art der Generierung von Code aus Modellen erläutern, die sogenannte metamodellbasierte Codegenererierung.

Metamodelle

Metamodelle beschreiben die Modellierungselemente, die im Rahmen eines Modellierungsansatzes verwendet werden können. So definiert das Metamodell der UML zum Beispiel die Elemente, die im Rahmen von UML-basierter Modellierung verwendet werden können: Klassen, Attribute, Operationen, Assoziationen, usw. Zur Beschreibung dieses UML Metamodells wird eine Meta-Metasprache verwendet, die sogenannte MOF.

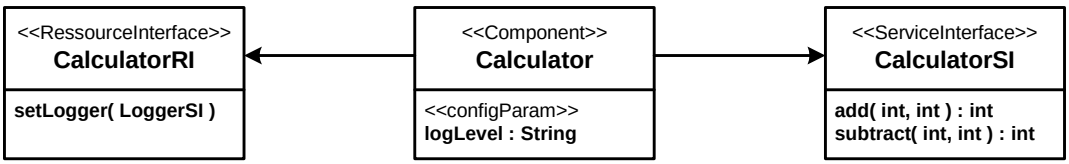
Eine Möglichkeit um Modelle bzgl. irgendwelcher Kriterien zu validieren ist daher, ein Metamodell zu definieren und Modelle (also Instanzen des Metamodells) gegen dieses zu validieren. Besonders nützlich ist dieser Ansatz dann, wenn das Metamodell vom Anwendungsprogrammierer selbst anpaßbar ist. In einem weiteren Schritt kann dann daraus ggfs. Implementierungscode generiert werden. Dieser Artikel soll anhand eines einfachen Beispiels zeigen, wie all dies praktisch funktioniert. Dabei kommt das b+m GeneratorFramework [BMWeb] zur Validierung von Modellen und zur Codegenerierung zum Einsatz. Modelliert wird mit einem beliebigen UML Werkzeug welches in der Lage ist, XMI zu erzeugen. Ich verwende hier MagicDraw 6 [magicDraw].

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

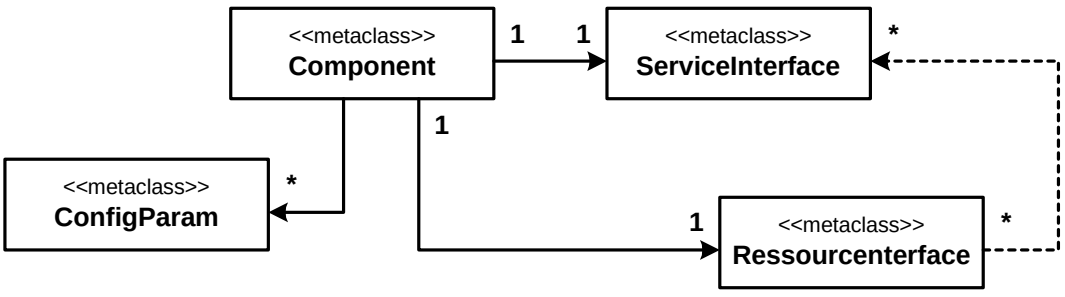
Architekturdefinition anzugeben, was eine Komponente ist. Wir beginnen also mit der Definition eines Metamodells für Komponenten dieser Infrastruktur.

Zunächst wollen wir zum Ausdruck bringen, daß eine Komponente ein Serviceinterface enthält 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 Ressourceinterface 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 Komponenten 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.

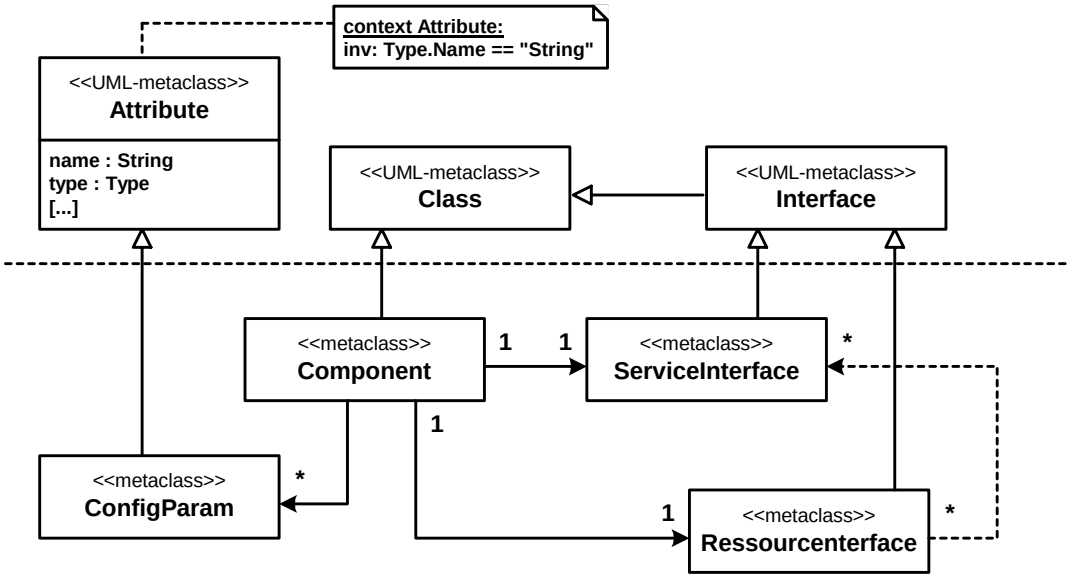


Es bringt zum formell Ausdruck, was wir oben verbal beschrieben haben – zumindest zum Teil. Die Ankopplung eines Instanzdiagrammes (der *Calculator*) an das Metamodell passiert mittels Stereotypen: Wenn wir zum Ausdruck bringen wollen, daß z.B. *logLevel* eine Instanz des Metatyps *ConfigParam* ist, 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 Modellverifizierer 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, das 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 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 verwendet 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 instantiieren. Hier das Beispiel für den *ConfigParam*:

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

Über eine (hier nicht gezeigte) Konfigrationsdatei 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* instantiieren soll. Aus Sicht des Generators ist die 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).

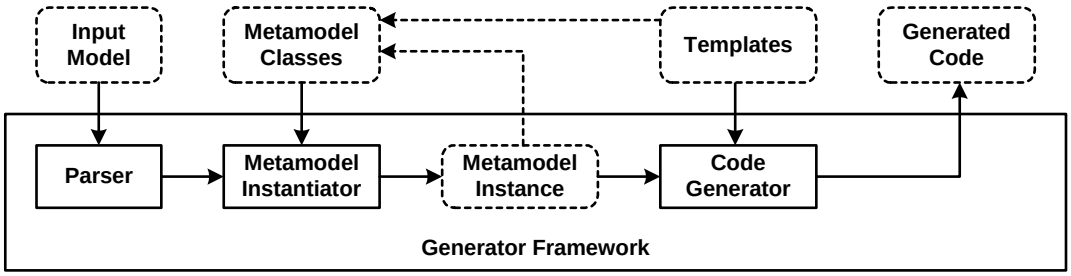
Nun bietet uns diese Klasse ConfigParam noch nicht viel Nutzwert. Insbesondere fehlt noch die Constraint, daß der Typ von *ConfigParams* immer *String* sein muß. Um derartige Constraints zu überprüfen besitzen alle Klassen des Metamodells eine Operation *CheckConstraints* die vom Generator aufgerufen 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 *CheckConstraint*s-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 Model. 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»
«ENDDEFINE»
```

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»
«ENDDEFINE»
```

Der zu generierende Code steht dabei einfach außerhalb der französischen Anführungszeichen. Diese dienen auch als Escape-Character um innerhalb des zu genererierenden 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»;
«ENDDEFINE»
```

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;
    }
«ENDDEFINE»
```

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 Metalasse *ConfigParam* an. Dies ist standardmäßig in der Metaklasse Attribut 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);
```

```
    }
  }
```

Zusammenfassung

Dieser Artikel konnte natürlich nur einen Überblick über die Metamodellbasierte Codegenerierung geben, und hoffentlich Appettit machen. Die Benefits eines solchen Ansatzes kommen natürlich vor allem bei Systemen hinreichend großer Komplexität zum Tragen. Auch gibt's bei der Verwendung des Generatorframeworks noch ein paar Details zu beachten; siehe [BMWeb]. Nichts desto trotz ist der hier beschriebene Ansatz zur Codegenerierung ein sehr mächtiger und es macht Spaß damit zu arbeiten. Ausprobieren!

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