Codegenerierung – ein Überblick

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

Dieser Artikel erläutert wann und wo Codegenerierung sinnvoll eingesetzt werden kann und zeigt einige weit verbreitete Technologien für Codegenerierung auf.

Wann ist Codegenerierung sinnvoll?

Codegenerierung ist ein wichtiges Mittel die Effizienz von Softwareentwicklungsprojekten zu steigern. Einen Überblick darüber, wann und wo Codegenerierung üblicherweise eingesetzt wird findet sich in [1]. Zusammenfassend kann man sagen, daß Codegenerierung in den folgenden Situationen sinnvoll einsetzbar ist:

- Das zu erstellende System (oder ein Mitglied einer Systemfamilie) muß eine große Flexibilität aufweisen, aus Performancegründen sind traditionelle, generische OO-Mittel allerdings nicht einsetzbar.
- Auch die Verringerung der Grösse des Programmimages kann ein Aspekt sein – der generierte Code wird dann nur das enthalten, was auch wirklich notwenig ist.
- Manchmal will man den Code auch einer statischen Analyse unterziehen (z.B. in der embedded Welt: Schedulability, Resourcenverbrauch, etc.). Generierter Code kann leichter analysiert werden als generischer Code.
- Ein anderer Grund für Codegenerierung (insbesondere aus Modellen) ist, daß Sie als Entwickler ihr Anwendung auf einem höheren Abstraktionslevel als dem von der Programmiersprache/Plattform angebotenen "programmieren" möchten. Dies ist insbesondere im Zusammenhang mit Produktfamilien und domänenspezifischen Modellen sinnvoll. Der Generator wird aus den abstrakten Modellen plattformspezifischen Implementierungscode generieren.
- Manchmal lassen sich bestimmte Dinge auch mit der Programmiersprache nicht ausdrücken (zum Beispiel kann man in Java im Rahmen eines Downcasts die Zielklasse nicht als Variable angeben). Generierter Code kann solche Dinge natürlich tun.
- Ein weiterer Grund für Generierung kann sein, dass Aspekte in das System eingebaut werden sollen. Dies kann, wie bei AspectJ, auf Quellcodeebene durch Codegenerierung ("Verweben") erfolgen.

Natürlich muß bei der Verwendung von Codegenerierung immer beachtet werden, dass der Generator erstellt (oder ein vorhandener parametriert) werden muß. Dies bedeutet Aufwand, bei dem man sich überlegen muß ob er sich lohnt. Üblicherweise lohnt er sich

dann, wenn der erstellte Generator mehrmals verwendet werden kann – also im Rahmen einer Softwaresystemfamilie.

Integration von generiertem und nicht-generiertem Code

Bevor wir und die Techniken und Tools zur Generierung von Quellcode anschauen, sollten wir uns noch der Frage widmen wie man generierten und nicht-generierten Code sinnvoll (also effizient und wartungsfreundlich) integriert. Man hört ja oft Aussagen wie „Ich finde Codegenerierung schlecht weil ich den generierten Code nicht verstehe und daher nicht modifizieren kann“. Das ist üblicherweise eine korrekte Aussage insofern, daß man generierten Code nicht immer versteht. Dies sollte allerdings auch gar nicht notwendig sein! Insbesondere sollte man sich zur Regel machen, generierten Code nie anzufassen und zu modifizieren (auch wenn dies bei vielen Tools durch sog. „protected areas“ möglich ist, also Codeteile, die beim nächsten Generierungslauf nicht überschrieben werden). Statt dessen sollte man sich auf die bewährten Techniken der objektorientierten Programmierung und einige der GoF Patterns besinnen. Der folgende Abschnitt erläutert, wie dies erreicht werden kann.

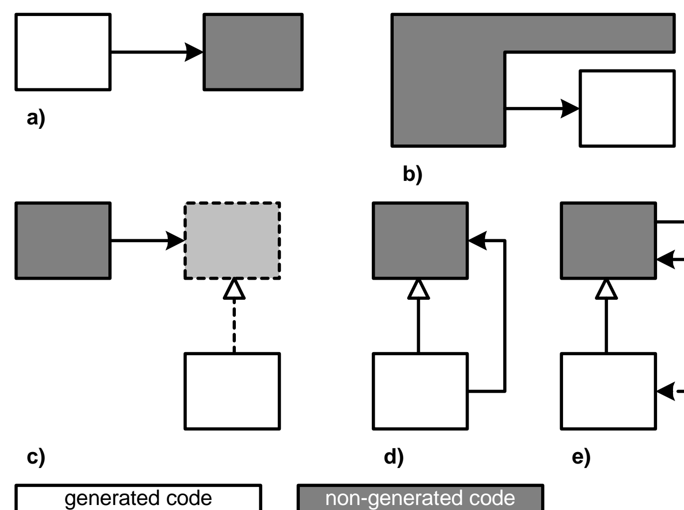

Abbildung 1: Möglichkeiten zur Integration von generiertem und nicht-generiertem Code

Der einfachste Fall (a) ist sicherlich, dass generierter Code nicht-generierten Code aufrufen kann. Das bedeutet im Klartext, daß man immer nur soviel Code generieren sollte wie unbedingt nötig: wenn möglich sollte der generierte Code auf soviel „normalen“ Code zugreifen wie möglich.

Natürlich geht dies auch umgekehrt (b): Nicht-generierter Code (ein Framework) kann auf generierten Code zugreifen. Um dies zu vereinfachen, bietet es sich oft an, ein Interface oder eine Abstrakte Klassen manuell zu entwickeln und den nicht-generierten Code gegen

diese Schnittstelle zu programmieren. Der generierte Code kann dann das betreffende Interface implementieren bzw. von der abstrakten Klasse erben. Factories (siehe GoF) können dabei helfen, generierten Code „einzupluggen"; siehe (c).

Eine Variation dieses Vorgehens ist es, daß generierte Klassen von einer nicht-generierten Klasse erbt und auf in der Basisklasse definierten Methoden zugreift (d). Auch andersherum ist es möglich: Ein Basisklasse kann abstrakte Methoden aufrufen die dann von einer konkreten, generierten Unterklassen implementiert werden (dies ist das *Template Method* Pattern aus GoF).

Beliebige Kombinationen dieser Vorgehensweisen sind natürlich möglich.

Generierungstechniken

Wenn man sich vor Augen führt in welchen Phasen der Programmerstellung Generierung sinnvoll und möglich ist, kann man die folgenden Phasen unterscheiden:

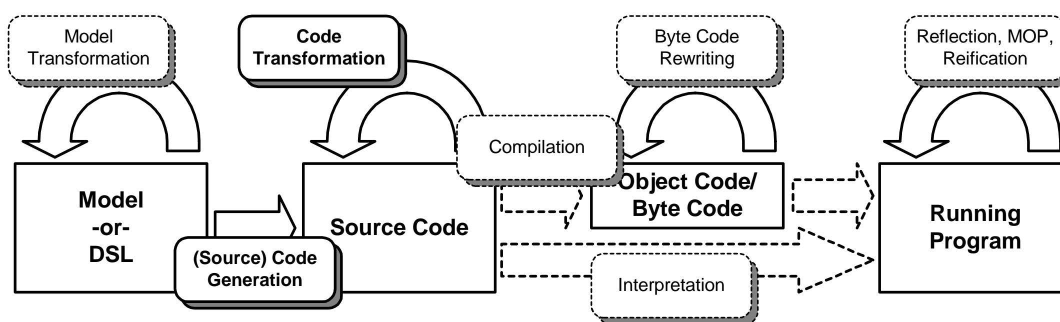

Abbildung 2: Die Phasen der Programmentwicklung

- Modeltransformation beschreibt die Transformation eines Modells in ein anderes (wobei wir hier einfach mal davon ausgehen, daß wir traditionellen Programmcode nicht als Model ansehen).
- Generierung von Programmcode aus Modellen ist sicherlich die verbreitetste Anwendung von Codegenerierung.
- Man kann auch bereits bestehenden Quellcode transformieren oder mit anderem bestehendem Quellcode verweben. AspectJ ist ein Beispiel.
- Compilieren kennen wir ja nun alle und wird deshalb hier nicht mehr weiter betrachtet.
- Man kann auch den compilierten Code nochmals transformieren. Dies wird meist bei Plattformen gemacht, die nicht direkt Maschinensprache sondern irgend eine Art von Zwischencode verwenden. Dies findet man z.B. oft im Zusammenhang mit objektorientierten Datenbanken.

- Auch zur Laufzeit kann Code generiert werden, üblicherweise direkt ausführbarer Bytecode.

Templates + Filtering

Diese Generierungstechnik beschreibt den wohl einfachsten Fall von Codegenerierung. Mit Hilfe von Templates wird über die relevanten Teile eines textuell repräsentierten Models iteriert (z.B. mit XSLT über XML). Der zu generierende Code steht in den Templates. Variablen in den Templates können an Werte aus dem Modell gebunden werden. Im folgenden ein einfaches Beispiel, wo eine Personen-JavaBean aus einer XML Spezifikation generiert wird (der Einfachheit halber verwenden wir hier nicht XMI...). Hier die Spezifikation:

```
<class name="Person" package="de.voelter.test">
  <attribute name="name" type="String"/>
  <attribute name="age" type="int"/>
</class>
```

Der generierte Code soll z.B. folgendermaßen aussehen:

```
package de.voelter.test;
public class Person {
  private String name;
  private int age;
  public String get_name() {return name;}
  public void set_name( String name) {this.name = name;}
  public int get_age() {return age;}
  public void set_age( int age ) {this.age = age;}
}
```

Das XSLT Stylesheet um dieses zu erreichen sieht dabei ungefähr aus wie das folgende Beispiel:

```
<xsl:template match="/class">
  package <xsl:value-of select="@package"/>;
  public class <xsl:value-of select="@name"/> {
    <xsl:apply-templates select="attribute"/>
  }
</xsl:template>

<xsl:template match="attribute">
  private <xsl:value-of select="@type"/>
    <xsl:value-of select="@name"/>;
  public <xsl:value-of select="@type"/>
    get_<xsl:value-of select="@name"/>() {
       return <xsl:value-of select="@name"/>;}
  public void set_<xsl:value-of select="@name"/> (
    <xsl:value-of select="@type"/>
     <xsl:value-of select="@name"/>) {
       this.<xsl:value-of select="@name"/> =
         <xsl:value-of select="@name"/>;  }
```

```
</xsl:template>
```

Das Generieren mittels Templates+Filtering funktioniert recht einfach und portabel, allerdings werden die Stylesheets schnell unübersichtlich und komplex. Für größere Systeme ist dieser Ansatz daher eher ungeeignet.

Templates + Metamodel

Um die Probleme mit der direkten Generierung aus Code aus (XML-) Modellen zu umgehen, kann man einen mehrstufigen Generator implementieren, der zunächst das XML parst, dann ein (vom Benutzer anpaßbares) Metamodel instantiiert und dieses dann zusammen mit Templates zur Generierung verwendet. Die folgende Abbildung illustriert dies.

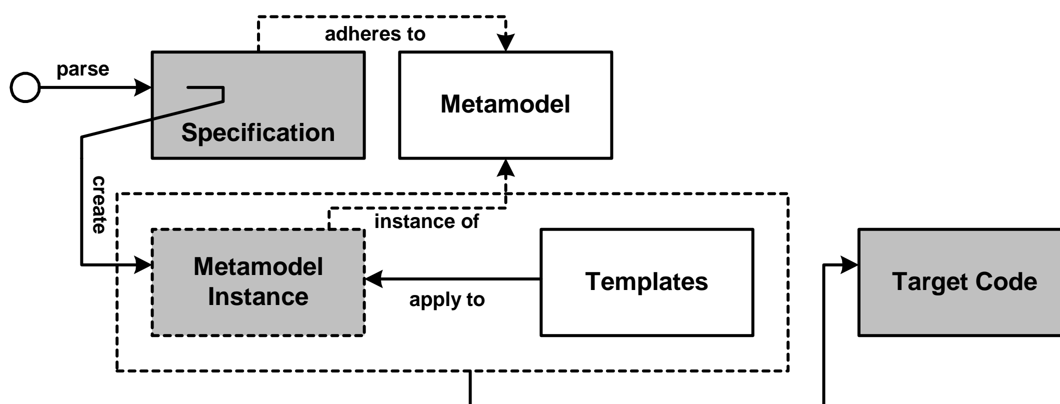

Abbildung 3: Der Ablauf der Generierung unter Verwendung von Templates+Metamodel

Der Vorteil dieses Vorgehens ist, daß man dadurch zum einen unabhängiger von dem konkreten Format des Modells wird (z.B. den verschiedenen XMI Versionen...). Zum anderen kann komplexere Logik zur Verifikation des Models im Metamodel untergebracht werden. Dies kann – im Gegensatz zu den Templates – in einer „richtigen" Programmiersprache, also z.B. Java, implementiert werden.

Ein Beispiel: Angenommen, wir wollen im Rahmen einer Komponentenarchitektur festlegen, daß jede *Komponente* eine Menge von *Konfigurationsparametern* hat und daß diese Parameter per Definition vom Typ *String* sein müssen (z.B. weil sie aus einem *.properties* File ausgelesen werden sollen). Man kann dann eine Metaklasse *Component* als Unterklasse der Standard-MOF-Metaklasse *Class* definieren, die eine Menge solcher Parameter besitzt. Die Parameter sind eine Unterklasse der Metaklasse *Attribute*. Die Überprüfung der Bedingung daß alle Konfigurationsparameter Strings sein müssen, kann dann z.B. folgendermaßen implementiert werden:

```
public class ConfigParam extends Attribute {
  public void checkConstraint() {
```

```
    if ( getType() != Type.STRING ) {
      throw new DesignError(„ConfigParams have to be Strings");
} } }
```

Die Generierungstemplates können dann mit Konzepten des Metamodells umgehen und außerdem davon ausgehen, dass die Modelle konsistent sind – sonst würde ja vom Generator *vor* Ausführung der Templates ein Modellierungsfehler gemeldet. Die folgende Template definiert eine Klasse pro Komponente im Model und erzeugt einen Setter für jeden Konfigurationsparameter:

```
<<FOREACH Component AS c IN Model {>>
  public class <<c.Name>> extends ComponentBase {
    <<FOREACH ConfigParam AS p {>>
      private String <<p.Name>>;
        public void configure_<<p.Name>>( String p ) {
        this.<<p.Name>> = p;
      }
    <<}>>
  }
<<}>>
```

API basierte Generatoren

Die wahrscheinlich wohlbekannteste Art von Codegeneratoren sind die API basierten. Diese stellen einfach eine API zur Verfügung, mit der Elemente der Zielplattform oder Sprache erzeugt werden können. Im folgenden zur Abwechslung mal ein Beispiel aus der Welt von .NET; der folgende Code soll erzeugt werden:

```
public class Vehicle : object {
}
```

Das folgende Stückchen C# Code erzeugt obigen:

```
CodeNamespace n = ...
CodeTypeDeclaration c = new CodeTypeDeclaration („Vehicle");
c.IsClass = true;
c.BaseTypes.Add (typeof (System.Object) );
c.TypeAttributes  = TypeAttributes.Public;
n.Types.Add(c);
```

Obiger Code baut eine interne Repräsentation des Codes auf, typischerweise als abstrakter Syntaxbaum (AST). Ein Aufruf einer entsprechenden Hilfsfunktion gibt dann den wirklichen Quellcode aus.

Diese Art von Generatoren ist recht intuitiv und einfach zu verwenden und es ist dabei auch recht einfach zur Erzwingen, daß nur syntaktisch korrekter Code erzeugt werden kann: der Compiler des Generatorcodes in Kombination mit der API kann dies sicherstellen. Allerdings existiert hier das Problem, daß man große Mengen von immer gleichem Code auch „programmieren" muß, statt einfach Template zu definieren, in denen ein paar Textstellen ersetzt werden.

Für Java existieren hierfür Tools die auf Quellcodeebene oder auf Bytecodeebene arbeiten. Details siehe [2].

Inline Generierung

Inline Generierung beschreibt den Fall wo im regulären Quellcode Konstrukte enthalten sind, die während der Compilierung weiteren Quell- oder Byte/Maschinencode erzeugen. Beispiele sind C++ Präprozessoranweisungen oder C++ Templates. Für Java existiert kein solcher Mechanismus[1], daher soll hier auch nicht weiter darauf eingegangen werden.

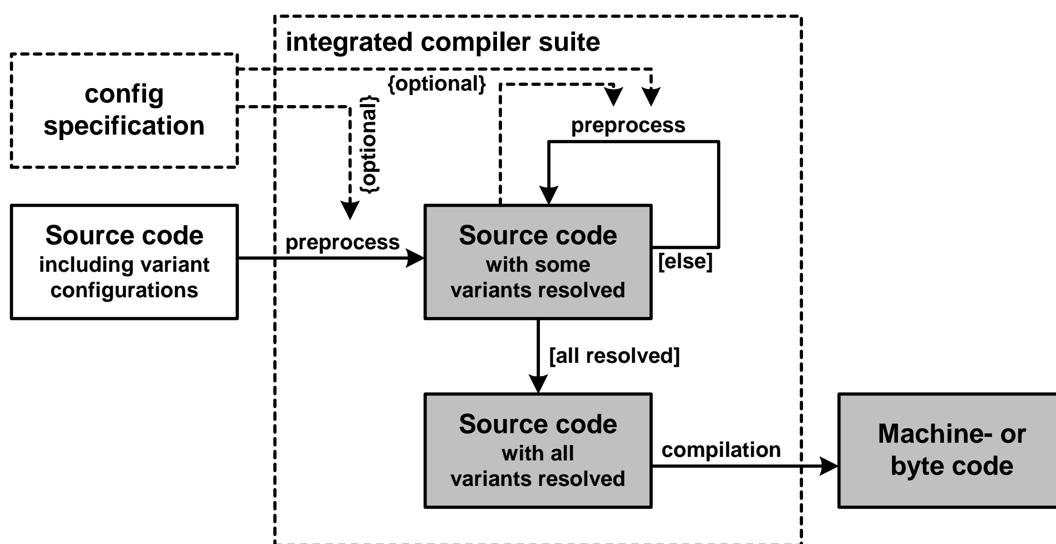

Abbildung 4: Das Vorgehen bei Inline-Generierung

Code Weaving

Code Weaving beschreibt das Zusammenfügen verschiedener, jeweils vollständiger und daher unabhängiger Codebestandteile. Dazu muß definiert werden wie diese verschiedenen Bestandteile zusammengewoben werden können – man nennt diese Stellen auch Join Points. Ein Beispiel für diese Art von Generator ist AspectJ. Hier werden regulärer OO Programmcode und sogenannte Aspekte auf Quellcode- oder Bytecodeebene verwoben. Aspekte beschreiben dabei sogenannte Querschnittliche Belange (engl. crosscutting concerns), also solche Funktionalität, die mit regulären OO Mitteln nicht schön lokalisiert (in einer Klasse oder einer Funktion) beschrieben werden können.

Im Folgenden als Beispiel ein Aspekt, der bei allen Klassen die irgendwelche Methoden auf der Klasse *Account* aufrufen Logausgaben einbaut. Vor jedem Methodenaufruf wird auf

[1] Auch die Generics die mit Java 1.5 kommen werden funktionieren nicht auf diese Weise.

die Konsole ausgegeben, *aus welcher Methode heraus* die Methode auf einem Account aufgerufen wird:

```
aspect Logger {
 public void log( String className, String methodName ) {
   System.out.println( className+".“+methodName );
 }
 pointcut accountCall():  call(* Account.*(*));
 before() calling: accountCall() {
   log( thisClass.getName(), thisMethod.getName() );
 }
}
```

Nachdem nun dieser Aspekt auf ein System angewandt wurde (also durch den Weaver mit dem regulären Code zusammengewoben wurde) wird ungefähr folgender Code erzeugt:

```
public class SomeClass {
  private Account someAccount = ...;
  public someMethod( Account account2, int d ) {
    System.out.println(„SomeClass.someMethod“); // aspect Logger
    someAccount.add( d );
    System.out.println(„SomeClass.someMethod“); // aspect Logger
    account2.subtract( d );
  }
  public void anotherMethod() {
    System.out.println(„SomeClass.anotherMethod“);
                                          //aspect Logger
    int bal = someAccount.getBalance();
  }
}
```

Codeattribute

Zu guter Letzt sei noch ein weiterer Mechanismus erwähnt der im Java Umfeld recht verbreitet ist: Codeattribute. Angefangen hat dies im Java-Umfeld mit JavaDoc, wo spezielle Kommentare verwendet wurden, um die automatische Generierung von HTML-Dokumentation zu erlauben. Aufgrund der erweiterbaren Architektur von JavaDoc kann man nun eigene Tags und Codegeneratoren einhängen. Das wohl bekannteste Beispiel dafür ist XDoclet. Hiermit werden unter anderem EJB Remote/Localinterfaces sowie Deployment Descriptoren generiert. Der Entwickler schreibt dabei nur noch die Implementierungsklasse von Hand, und fügt dieser Klasse eben entsprechende Kommentare dazu, die vom Code-Generator ausgelesen werden. Der Generator hat außerdem Zugriff auf den Syntaxbaum des Quellcodes an den die Kommentare angefügt sind. Damit kann der Generator Informationen sowohl aus den Kommentaren als auch aus dem Code selber entnehmen. Das folgende ist ein Beispiel einer Javaklasse der mit XDoclet-Kommentaren ergänzt wurde:

```
/**
 * @ejb:bean type="Stateless"
 *           name="vvm/VVMQuery"
```

```
 *            local-jndi-name="/ejb/vvm/VVMQueryLocal"
 *            jndi-name="/ejb/vvm/VVMQueryRemote"
 *            view-type="both"
 */
public abstract class VVMQueryBean
    /**
     * @ejb:interface-method view-type="both"
     */
    public List getPartsForVehicle( VIN theVehicle ) {
        return super.getPartsForVehicle( theVehicle );
    }
}
```

Das charmante an dieser Methode der Codegenerierung ist daß eben viele der Informationen die der Generator benötigt bereits im Code vorhanden sind. Der Entwickler muß nur noch wenig spezielle Kommentare angeben.

Zusammenfassung

Dieser Artikel konnte nur einen Überblick über Codegenerierungstechniken geben. Eine ausführlichere (englische) Beschreibung in Form von Patterns finden sich unter [2]. Ausserdem finden sich unter [3] eine Menge von Folien die sehr viele Beispiele enthalten. Beide enthalten auch Hinweise auf konkrete Tools incl. der URLs wo man sie herbekommt.

Referenzen

[1] Völter, Stahl; *Architektur und Generierung*; iX 03/2003

[2] Markus Völter; *A Catalog of Patterns for Code Generation;* www.voelter.de/data/pub/ProgramGeneration.pdf

[3] Markus Völter; Code Generation: A survey of Concepts, Techniques and Tools; www.voelter.de/data/presentations/ProgramGeneration.zip