Architektur und Generierung

Markus Völter, voelter@acm.org, www.voelter.de
Thomas Stahl, t.stahl@bmiag.de

Einführung

Generierung von Quellcode im Rahmen von Entwicklungsprojekten gilt heutzutage als akzeptiertes Konzept zur Effizienzsteigerung innerhalb eines Projektes, nicht zuletzt durch die zunehmende Verbreitung generativer Programmiertechniken (GP, [EC00]) und der Model-Driven Architecture (MDA™, [MDA]) der OMG. Mittlerweile gibt es auch die ersten praxistauglichen Tools auf dem Markt, die flexibel und ausgereift genug sind, die genannten Konzepte zu unterstützen. Allerdings ist dabei wie immer zu beachten, dass die Tool-Frage nicht im Vordergrund stehen sollte, sondern die Anforderungen. Ein Werkzeug ist eben dann geeignet, wenn es die Anforderungen erfüllt. So nähern wir uns der Frage nach den Voraussetzungen für den effektiven Einsatz von Codegenerierung.

Voraussetzungen für den Einsatz von Codegenerierung

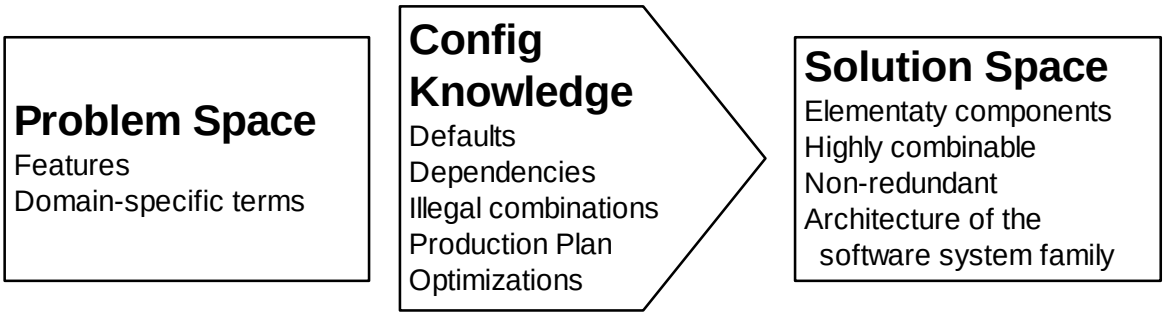
Ganz allgemein betrachtet lässt sich hier schnell eine Antwort finden: Codegenerierung ist immer eine systematische Konkretisierung oder Makroexpansion. Es muss also irgendeine abstrakte Spezifikation geben, so dass eine automatische Transformation in das gewünschte konkrete Ziel möglich ist. Dies ist natürlich ein altbewährtes Prinzip mit einer langen Tradition in der Informatik. Jede compilierte Hochsprache basiert auf diesem Paradigma. Durch MDA und GP vollzieht sich gerade der nächste Evolutionsschritt, denn der Abstraktionsgrad wird um eine Stufe angehoben. Spezifiziert wird nicht mehr in Form von Programmiersprachen sondern in Form von semantisch angereicherten Modellen. Diese können grafisch oder textuell notiert sein. Je mehr Bedeutung in den verwendeten Modellelementen selbst bereits steckt, desto größer ist der Nutzen: Die Modelle werden dadurch kompakter und die Arbeitserleichterung durch die automatisierte Transformation steigt. Um das aber erreichen zu können sind einige Dinge notwendig bzw. sinnvoll:

- Die zur Verfügung stehenden Basiskonstrukte müssen wohl definiert sein[1]. Dies bedeutet, dass sowohl ihre Syntax als auch die statische Semantik z.B. mit Hilfe eines sogenannten Metamodelles festgelegt werden muss.
- Die Transformationen der Basiskonstrukte müssen so definiert sein, dass sie maschinell umsetzbar sind.

[1] Im Falle von MDA geschieht dies durch ein sogenanntes UML-Profil - bestehend aus Stereotypen, *Tagged Values* und *Constraints*. Bei GP kommen Merkmal-orientierte, domänenspezifische Modellierungssprachen (DSL – Domain Specific Languages) zum Einsatz

- Die Basiskonstrukte sollten so gewählt sein, dass sie einerseits möglichst universell einsetzbar und andererseits möglichst redundanzfrei sind. Ein Satz von Konstrukten wird immer nur eine gewisse Klasse von Anwendungen, Systemen oder Aspekten beschreiben können. Die Basiskonstrukte sind dabei üblicherweise architektonisch oder fachlich motiviert.

Die generative Programmierung schematisiert den Transformationsvorgang mit Hilfe des sogenannten generativen Domänenmodells [EC00]:



In dieser Terminologie spannen die Basiskonstrukte den Problemraum auf, die Transformationen sind Teil des Konfigurationswissens und das Generat liegt im Lösungsraum.

Für die Praxis in einem Software-Projekt lässt sich folgendes ableiten:

- Die Basiskonstrukte müssen zum Projekt passen, d.h. es sind benannte, technische oder fachliche Konzepte der zu erstellenden Software, die schematisch umsetzbar sind.
- Die Transformation muss gleichermaßen zum Projekt passen und z.B. die gewünschte Zielplattform oder (Programmier-)Sprache bedienen.

Trennung in funktionale und technische Belange

Damit sind wir schon bei ganz konkreten Problemstellungen, denn ganz offensichtlich ergeben sich insbesondere aus dem ersten Punkt Anforderungen an die Struktur der Anwendung. Es wird eine sorgfältig definierte Anwendungsarchitektur benötigt, die einen optimal schematisierten Code zuläßt bzw. impliziert. Zunächst wäre da die Trennung eines Systems in technische und funktionale Aspekte. Funktionale Aspekte sind die Anteile eines Systems, die die fachlichen Anforderungen realisieren. Technische Aspekte sind all die Dinge, die sich mit den nicht-funktionalen Anforderungen befassen. Das klassische Beispiel: In einer Internet-Shopping-Anwendung könnten die funktionalen Anforderungen z.B. Dinge sein wie durchsuchen des Katalogs, Verwaltung des Warenkorbs, oder Durchführen einer Bestellung. Technische Aspekte wären dann z.B. Dinge wie

transaktionale Absicherung des Bestellvorgangs, Speicherung der Daten (Persistenz) und die Ausfallsicherheit und Skalierbarbarkeitsanforderungen des Gesamtsystems.

Üblicherweise sind bei verschiedenen Systemen innerhalb einer Domäne die technischen Anforderungen sehr ähnlich: Transaktionen, Security, oder Lastverteilung und Failover finden sich bei fast allen e-Business Anwendungen. Für DRE-Systeme (Distributed, Realtime Embedded-Systeme) sind die technischen Aspekte Dinge wie Prioritätskontrolle, verteiltes Resourcenmanagement oder Scheduling.

Idealerweise kann man nun die technischen Aspekte aus der Anwendung auslagern, indem man z.B. Frameworks verwendet um die technischen Aspekte zu realisieren. Die funktionalen Anteile werden dann in dieses Framework „eingesteckt“ und während der Ausführung des Programms vom Framework verwendet (siehe Abb 1).

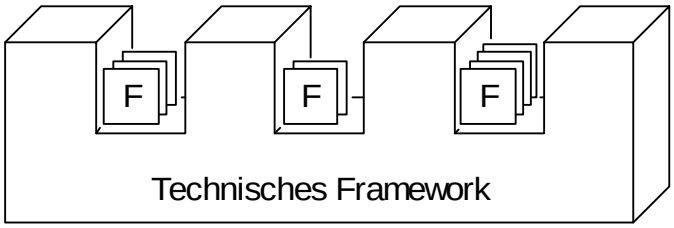


Abbildung 1

Damit offenbart sich bereits eine gute Gelegenheit für Codegenerierung: Und zwar in der Zwischenschicht zwischen den funktionalen Bausteinen und dem technischen Framework. Der Grund ist folgender: In vielen Fällen wird das Framework relativ generisch sein. Es soll also mit praktisch beliebigen funktionalen Bausteinen zurecht kommen können und sich damit für eine möglichst breite Anwendungspalette verwenden lassen. Es ist aber oft aus Effizienzgründen oder auf Grund von Einschränkungen der Programmiersprache (i.d.R. deren Typsystem) nicht möglich, beliebigen funktionalen Code in solch ein Framework zu integrieren. Man verwendet dann oft einen sogenannten Glue-Code-Layer, eine „Leimschicht“, die den funktionalen Code an das Framework adaptiert (siehe Abb 2).

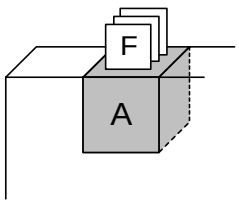


Abbildung 2

Beispiele

Da diese Schicht üblicherweise sehr schematisch aufgebaut ist, läßt sich solcher Code leicht generieren. Einige Beispiele dazu.

- CORBA-Object Request Broker (ORBs) sind Frameworks zur Remote Kommunikation. Sie erlauben es, Objekte mit beliebigen (in IDL definierbaren) Interfaces entfernt über Rechnergrenzen hinweg anzusprechen. Das Framework kann sehr effizient Methodenaufrufdaten über Rechnergrenzen hinweg transportieren. Der Programmierer spezifiziert die IDL-Interfaces und implementiert die Programmlogik in passenden Implementierungsklassen. Der generierte Code in den Stubs macht aus den Methodenaufrufen des Clients Datenpakete, die das Framework dann transportiert. Da die Typen (zur Codegenerierungszeit) bekannt sind, kann der nötige Code sehr einfach generiert werden. Gleiches gilt für die serverseitigen Adapter-Klassen, welche die Aufrufe an die fachlichen Implementierungen weiterleiten.
- Enterprise JavaBeans sind eine Komponenteninfrastruktur [VSW02]. Hier liegen die technischen Aspekte in einem sogenannten Container der die den funktionalen Code enthaltenden Komponenten ausführt. Er kümmert sich um Transaktionen, Security, Naming etc. Auch hier wird eine Glue-Code Schicht generiert, die sich anhand der Spezifikationen im Deployment Deskriptor um die Adaption der Komponenten an den Container kümmert.

Wie am Beispiel EJB gut sichtbar wird, benötigt man, um Code generieren zu können, immer eine Beschreibung der Anfoderungen, was der Code tun soll – d.h. ein Modell. Im Fall von CORBA ist dies so einfach, dass das *IDL-Interface* des zu adaptierenden Objektes ausreicht. Im Falle von EJB reicht dies nicht: Es werden weitere Angaben benötigt, die hier in Form von XML-Dateien, den Deployment Deskriptoren, gemacht werden. Man beachte, dass diese XML-Dateien selbst keinen „programmierten Code" darstellen. Es handelt sich lediglich um eine Spezifikation des erwünschten Verhaltens. Eine weit verbreitete Möglichkeit strukturelle Aspekte zu spezifizieren, sind UML Diagramme, insbesondere Klassendiagramme. Zur Weiterverarbeitung in Generatoren werden diese Modelle dann üblicherweise als XMI-Dateien [XMI] exportiert, einem standardisierten, auf XML basierenden Format zur Beschreibung von UML Modellen[2].

Codegenerierung und Aspektorientierung

Aspekte sind Funktionalitäten eines Systems, die sich nicht ohne weiteres an einer Stelle im Code (z.B. einer Komponente, einer Klasse, einer Operation) lokalisieren lassen, sondern „quer" durch das ganze System verteilt sind. Klassische Beispiele sind Fehlerbehandlung,

[2] Genau genommen können mit XMI nicht nur UML Modelle serialisiert werden, sondern alle Modellierungssprachen, die auf der Meta Object Facility (MOF) Spezifikation basieren.

Threading und Synchronisierung, Persistenz, Transaktionen, Security usw. Offensichtlich wirft dieses „Querliegen" ein Problem bzgl. Modularisierung und damit der Wartbarkeit auf. Das Einfügen, Modifizieren oder Entfernen eines solchen Aspektes kann sehr aufwendig sein, da potentiell an sehr vielen Stellen im Code modifiziert werden muss. Die Aspektorientierte Programmierung (AOP) versucht nun (z.B. durch neue Sprachkonstrukte) diese querliegenden Aspekte in separaten Modulen, den Aspekten, zu lokalisieren.

Die Abstraktionsleistung steht bei diesem Ansatz zunächst nicht im Vordergrund, d.h. es handelt sich eigentlich nicht um eine Codegenerierung im hier definierten Sinne, sondern eher um eine Maßnahme zur Modularisierung von Sourcecode: AOP-Tools, wie z.B. AspectJ verwenden aspektfreien Sourcecode sowie den separaten Aspektcode als Eingabe, um in der Ausgabe beides über sogenannte *Join Points* zu „verweben".

Dennoch ist die aspektorientierte Sichtweise auch für ‚echte' Codegenerierung (in unserem Sinne) relevant. Wichtig ist hier wiederum eine Anwendungsarchitektur, welche die zu generierenden Aspekte erst einmal explizit definiert. Auf dieser Grundlage können dann nämlich auch modellbasierte Transformationen definiert werden, die gerade einzelne Aspekte abdecken. Ein Beispiel: Angenommen, wir haben einen Generator der aus UML-Klassendiagrammen Java-Klassen generiert. Und angenommen, wir wollen allen Klassen eine zusätzliche Operation spendieren, oder am Anfang aller Operationen einen Log-Eintrag ausgeben. Dann ist dieser Code im Generator idealerweise an genau *einer* Stelle zu finden – im einfachsten Falle dort, wo Klassen bzw. Operationen generiert werden. Damit ist ein Aspekt des Zielsystems an *einer* Stelle im Generator realisiert[3] und man kann den Aspekt z.B. durch einen einfachen Konfigurationsschalter in das Generat einfügen lassen oder komplett ausblenden.

Die Vorteile eines Aspect-Weavers (Modularisierung von Aspekten und Konfigurierbarkeit) lassen sich so auf einen modellbasierten Codegenerator übertragen.

Generierung vs Generizität

Die bisher genannten Anwendungen der Code-Generierung sind sicherlich am weitesten verbreitet. Sie haben auch den Vorteil, dass der Entwickler üblicherweise nie in den generierten Code reinschauen, oder ihn womöglich sogar ändern muss – man hat also keine Round-Trip Problematik.

Im Extremfall kann aber auch die komplette Infrastruktur eines Systems auf Basis einer definierten Anwendungsarchitektur generiert werden. Gerade im eBusiness-Bereich gibt es hierfür bereits einige erfolgreiche Beispiele, wie [STA02]. Je elaborierter die Anwendungsarchitektur und das zugehörige Programmiermodell, desto schematischer ist

[3] Es ist natürlich nicht immer möglich, jeden Aspekt des Zielsystems als genau ein Modul im Generator zu realisieren.

der Sourcecode der Anwendungen aufgebaut. Dadurch lässt sich ein Großteil der Programmierung auf Copy&Paste-Tätigkeiten zurückführen, bei der im Wesentlichen nur fachliche Bezeichner ausgetauscht werden müssen. Genau diese Tätigkeiten sind nun in Form einer modellbasierten Transformation automatisierbar. Die Folge ist, dass die Implementierung jedes architektonischen Schemas, wie bei der Aspektorientierung, nur noch an *einer* Stelle im Generator definiert ist. Die schematischen Anteile werden gleichsam aus dem Anwendungs-Sourcecode herausfaktorisiert. Durch Generierung aus einem Modell entsteht dann umgekehrt ein entsprechender, architektonischer Implementierungsrahmen für die fachliche Anwendungsprogrammierung. Dass ein derartiges Vorgehen letztlich zu stark verbesserter Entwicklungseffizienz, Software-Qualität und Wartbarkeit führt, ist offensichtlich.

Ihre Stärke beziehen diese Ansätze insbesondere auch aus dem Zusammenspiel von generischen und generativen Techniken. Diese schließen sich keineswegs wechselseitig aus - im Gegenteil, zumeist ergänzen sie sich geradezu optimal. Abstrakte Basisklassen, technische Services und Frameworks können gemeinsam ein anwendungsneutrales Laufzeitsystem bilden, auf welches das Generat genau abgestimmt ist. Der Generator nutzt also Wissen um die Existenz und Beschaffenheit eines generischen Laufzeitsystems. Dies vermeidet Coderedundanz im Generat und verbessert die Wartbarkeit des Systems.

Die Frage, was generiert wird und was in Form von generischen Bibliotheken vom Generat verwendet wird, hängt von verschiedenen Faktoren ab:

- Performanz: Spezifischer, generierter Code ist üblicherweise effizienter als generische Bibliotheken, da zur Generierungs- und Compilierungszeit mehr Informationen bekannt sind.
- Codegröße: Generischer Code ist oft kleiner; durch wiederholte Expansion kann der erzeugte Quell- und Objektcode größer werden.
- Spracheigenschaften: Manche Sprachen haben inhärente Beschränkungen im Typsystem, bestimmte Dinge sind also einfach nicht ausdrückbar. Dort kann Codegenerierung helfen (Java hat im Gegensatz zu C++ z.B. keine Templates).
- Bestehende System- und Anwendungslandschaft: In den meisten Fällen ist bereits eine ganze Reihe wieder verwendbarer, generischer Komponenten (z.B. Application-Frameworks) im Umfeld etabliert, so dass es unsinnig wäre, sie durch generierten Code zu ersetzen.

Generierung von funktionalem Code

Bisher haben wir uns ausschliesslich mit der Generierung auf Seiten der technischen Infrastruktur beschäftigt. Man kann nun aber auch versuchen, die funktionalen Bausteine zu generieren. Um dies zu erreichen, muss man üblicherweise zunächst die Freiheiten des Entwicklers einschränken. Denn auch hier werden die generierten Artefakte üblicherweise

in einem technischen Framework zur Ausführung kommen, und dieses kann eben nicht alles können[4]. Auch wieder einige Beispiele:

- Die Convergent Architecture [HT01] kennt nur drei Kernkonzepte: Organisationen, Prozesse und Resourcen (man erkennt daran schon ungefähr, für welchen Bereich von Anwendungen dieser Ansatz geeignet ist). Der Entwickler kann diese Dinge mittels UML beschreiben (und durch spezielle Attribute näher charakterisieren) um dann daraus Code zu generieren, der bereits sehr viel „richtige" Funktionalität enthält. In Anlehnung an die MDA wird diese Funktionalität in „Schablonen" gegossen die dann auf Plattformen wie z.B. der J2EE ablaufen.
- Eine andere Metapher die vor allem in Workflowsystemen zum Einsatz kommt sind Prozesse und Aktivitäten, wobei ein Prozess eine Sammlung von Aktivitäten und deren Zusammenhänge und Abhängigkeiten darstellt. Man kann dann sehr gut grafisch Prozesse modellieren, die Transitionsbedingungen spezifizieren und den Datenfluss modellieren – ein Generator kann daraus dann ablauffähigen Code erzeugen.
- In der Entwicklung von embedded Systemen werden zur Spezifikation von diskretem Verhalten üblicherweise Zustandsdiagramme eingesetzt. Auch hier sind die Ausdrucksmittel des Entwicklers beschränkt: Zustände, deren Übergänge, Transitionsbedingungen und Aktionen, die bei Eintritt, Verlassen oder während des Verweilens in einem Zustand ausgeführt werden sollen. Auch daraus läßt sich sehr effizienter Code generieren der dann unter Kontrolle eines Framework (meist einem Echtzeitbetriebssystem) ausgeführt wird.
- Auch für nicht-disktrete Systeme im embedded Bereich werden abstrakte Spezifikationen eingesetzt: hier nun allerdings keine Zustandsdiagramme, sondern Funktionsbausteine die Integrieren, Summieren, oder Differenzieren können (d.h. mathematische Operationen abbilden). Durch Kombination solcher Bausteine und Beschreibung des Datenflusses lassen sich auch hier Spezifikationen erzeugen, die sich sehr schön als Basis für die Codegenerierung eignen.

Kategorien von Codegeneratoren

Es gibt, wie bereits ersichtlich, verschiedene Arten von Generatoren. Zum einen sind da die „geschlossenen", nicht-flexiblen Werkzeuge. Sie bieten vorgefertige – aber durchaus optimierte - Lösungen für ganz spezifische Problemstellungen, sind aber weder im Problemraum noch im Lösungsraum anpassbar, oder anders ausgedrückt: Sowohl das

[4] Im Übrigen ist die Begrenzung der Freiheit der Programmierer auf einige wesentlichen Abstraktionsarten auch ohne die Verwendung von Generierung eine gute Idee. In großen Projekten können Entwicklungsteams so deutlich effizienter arbeiten.

zugrunde liegende Metamodell als auch die Transformationen sind fest. Zur Spezifikation (Modellierung) werden üblicherweise proprietäre, grafische oder textuelle Notationen verwendet. Beispiele für derartige Systeme sind vor allem im embedded Bereich und bei klassischen 4GL Tools zu finden. Auch die oben erwähnten CORBA-IDL- und EJB-Compiler fallen in diese Kategorie.

Das andere extrem sind komplett offene Systeme die keine Vorgaben über den zu generierenden Code und dessen Struktur machen. Sie stellen dem Entwickler die grundlegenden Mechanismen für Codegenerierung zur Verfügung. Dazu zählen Template-basierte Systeme und Frameprozessoren genauso wie AST-basierte APIs (z.B. [BMI]). Durch hierarchische Verwendung der grundlegenden Mechanismen lassen sich damit auch recht effizient mächtige, domänenspezifische Generatoren bauen. Bei Werkzeugen dieser Kategorie sind sowohl das Metamodell als auch die Transformationen variabel, d.h. frei definierbar.

Der Mittelweg existiert natürlich auch: Systeme mit vorgegebener Modellierungssprache aber anpassbaren Generierungsvorschriften (z.B. ‚Genlets'), wie sie von einigen CASE-Tools angeboten werden Hier handelt es sich um Werkzeuge mit festem Metamodell aber variabler Transformation.

Zu beachten ist generell, dass domänenspezifische Generatoren selbst Programme sind (Metaprogramme), so dass sich auch auf dieser Ebene letztlich wieder die Fragen nach Struktur, Wartbarkeit und Modularisierung ergeben. Idealerweise kommen hier bekannte und bewährte Konzepte wie Objektorientierung und Komponentenbildung zum Einsatz. Hierfür bieten allerdings nur wenige der heute verfügbaren Tools tragfähige Ansätze.

Eine häufige Problemstellung bei der automatischen Erzeugung von Sourcecode ist auch die Notwendigkeit, das Generat um manuell programmierte Fachlogik ergänzen zu müssen. Hier haben die Werkzeuge ggf. die Aufgabe, diese Inhalte bei iterativer Generierung zu erhalten.

Organistorische Aspekte

Um generative Techniken in großen Projekten erfolgreich einsetzen zu können, sind neben den Anforderungen an Architektur und Tools natürlich auch betriebswirtschaftliche und organisatorische Aspekte zu bedenken:

- Das Erstellen eines spezifischen Codegenerators ist ein gewisser Trade-Off gegenüber der manuellen Programmierung. Der Einsatz von Tools (z.B. eine Template-Engine) kann diesen Aufwand minimieren aber nicht eliminieren. Einige generative Tools bringen schon gewisse Basistransformationen für gängige Plattformen mit, aber auch diese müssen in aller Regel noch angepasst werden, um einen optimalen Nutzen zu bringen. Letztlich muss ein Generator seine Kosten (Lizenzkosten plus Metaprogrammierung) durch seinen Nutzen (Generierung, Qualitätssteigerung, Effizienz in gleichgelagerten Folgeprojekten etc.) eben aufwiegen.
- Moderne, komplexe Architekturen erfordern Spezialisten. Eine Chance, welche die generative Softwareentwicklung bietet, ist die Trennung von Anwendungsprogrammierung und Metaprogrammierung. Die Experten können ihr Wissen in Form von Generatoren zur Verfügung stellen. Dies bedingt aber ggf. entsprechende Teamstrukturen, die unter anderem auch eine Rückkopplung zur Metaprogrammierung ermöglichen.

Fazit

Codegenerierung ist ein praxistaugliches Konzept. Sie erlaubt eine effiziente und qualitativ hochwertige Software-Entwicklung. Es gibt unterschiedliche Ansätze und Werkzeuge zur Codegenerierung mit jeweils unterschiedlichen Zielsetzungen. Das volle Potenzial lässt sich allerdings nur auf Basis einer wohl definierten Software-Architektur ausschöpfen.

Referenzen

[BMI] b+m Informatik AG, b+m Generator FrameWork www.architectureware.de

[EC00] Eisenecker, U. und Czarnecki, K., *Generative Programming*, Addison-Wesley 2000

[HT01] Hubert R. und Taylor D., *Convergent Architecture*, Wiley, 2001

[MDA] OMG, *Model Driven Architecture*, www.omg.org/mda

[STA02] Stahl T., Generative Softwareentwicklung in der Praxis, OBJEKTspektrum 1/02