Generatives Programmieren

Tammo van Lessen
Fakultät für Elektrotechnik und Informatik
Universität Stuttgart
vanto@xnap.org

Abstract

Generatives Programmieren soll durch die Automatisierung von Produktionsschritten den Rückstand den die Softwareentwicklung gegenüber der produzierenden Industrie hat, aufholen. Im Produktlinienkontext ist das Ziel, nach einem ausführlichen Domain Engineering, die eigentliche Produkterstellung auf einen Knopfdruck zu reduzieren. Dieses Dokument beschreibt den Prozess des generativen Programmierens, die dahinterstehende Idee, sowie den Aufbau und Einsatz von Software-Generatoren.

1. Einleitung

Zu Beginn der Ära der Softwareentwicklung waren die Anforderungen an die Flexibilität der Programme eher beschaulich. Das änderte sich schnell mit der enormen Verbreitung von PCs in alle nur erdenklichen Bereiche. So mussten sich auch die Methoden zur Entwicklung der Software ändern, um wachsenden Anforderungen und wachsender Projektgröße gerecht zu werden. Es wurden imperative Programmiersprachen entwickelt, um die Wartbarkeit der Quellcodes zu erhöhen, man begann die Anforderungen und analog die Implementierung zu strukturieren und konnte dank dieser modularen Programmierweise Programme in Komponenten mit geringer Kopplung zergliedern. Auch hierdurch verbesserte sich die Wartbarkeit und die Programmierarbeiten ließen sich besser parallelisieren. Durch diesen Schritt war es aber nun auch möglich, Module einmal zu entwickeln und in mehreren Softwareprodukten zu verwenden. Wiederverwendbarkeit ist das Schlüsselwort und ist heute einer der wichtigsten und kostensparendsten Faktoren in der Softwareentwicklung. Das war unter anderem ein Grund, die Programmierparadigmen erneut zu überarbeiten. Man entwickelte die objektorientierte Programmierung. Mit ihr ist es leichter, reale Probleme in die Programmcodeabstraktion abzubilden. Das erlaubt eine neue Art der Strukturierung die sich besser wiederverwenden lässt. Sinnvoll ausgewählte und implementierte Klassenbibliotheken ermöglichen heutzutage das Entwickeln und Pflegen ganzer Software-Produktlinien. Wichtig ist hierbei, dass sowohl mit Wiederverwendung gearbeitet wird, als auch bei der Entwicklung der Komponenten darauf geachtet wird, dass diese wiederverwendet werden können (Design for Reuse, Design with Reuse).

In der produzierenden Industrie gab es eine ähnliche Entwicklung. Auch wenn eine Analogie zur Automobil-Industrie nur sehr vorsichtig betrachtet werden kann, weil bei der Entwicklung der immateriellen Software stets andere Gesetze gelten, gibt es doch einige Parallelen, die auch einen Blick in die mögliche Zukunft der Softwareentwicklung gestatten. John Hall verwirklichte 1885 erstmals das Konzept der austauschbaren Teile - eine Parallele zu der modularen Softwareentwicklung. Dennoch war der Herstellungsprozess der Automobile langsam und somit sehr teuer. Das änderte sich erst 1901 als Ransom Olds das erste Fließband einführte und dieses 1913 in verbesserter Form von Henry Ford in den Fertigungsprozess von Automobilen implementiert wurde. Überträgt man das Bild des Fließbandes wieder auf die Softwaretechnik kann man eine Analogie zu der Produktlinien-Architektur herstellen. Mit diesem Schritt hat die Softwaretechnik jedoch nicht gleichgezogen, denn die Automobilindustrie ist schon einen Schritt weiter: Industrieroboter sind in der Lage vollautomatisch Kraftfahrzeuge zu fertigen und zwar nicht immer mit identischer Ausstattung, sondern so wie es der Kunde bei seiner Bestellung angegeben hat. Man versucht seit der Jahrtausendwende die Automatisierung von Arbeitschritten auch in die moderne Softwareentwicklung einzuführen, denn die Vorteile liegen auf der Hand.

2. Generatives Programmieren

2.1. Ziele

Die Praxis hat gezeigt, dass die Objektorientierung im Hinblick auf Wiederverwendbarkeit noch nicht der Weisheit letzter Schluss ist. Klassen sind zu klein, zu atomar um sie sinnvoll wiederverwenden zu können. Generatives

Programmieren versucht, größere Bausteine besser wiederverwendbar zu machen und diese dann automatisch an die speziellen Anforderungen anzupassen. Man erhofft sich dadurch, wie auch in der Industrie, eine kürzere Entwicklungszeit der zu entwickelnden Softwarekomponenten und gleichzeitig eine höhere Produktivität. Der Entwickler soll sich nicht mehr mit dem Problem auf der Quellcodeebene beschäftigen müssen, sondern nur noch bestimmte Features aktivieren, die der Generator dann implementiert.

Dadurch kann Software schneller und flexibler entwickelt werden. Neben einer Steigerung der Softwarequalität, die aus der automatischen Code-Generierung resultiert - ein Generator macht keine Trivialfehler - möchte man auch die immer komplexer werdenden Aufgaben mit geringem Aufwand beherrschbar machen.

Generative Programmierung eignet sich ganz besonders für Produktlinien, da die einzelnen Produkte dieser Linie die gleiche Anwendungsdomäne bedienen und deshalb eine ähnliche Funktionalität bieten.

2.2. Definition: Generatives Programmieren

"Generative Programming (GP) is a software paradigm based on modeling of software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary reuseable implementation components by means of configuration knowlegde." [2]

Die Kernaussage dieser Definition lässt sich leicht zusammen fassen:

- Statt ein bestimmtes Software Produkt von Grund auf zu erstellen, setzt GP den Fokus auf eine Familie von ähnlichen Produkten. Die Ähnlichkeit bezieht sich hier auf eine gemeinsame Anwendungsdomäne.
- Der Kunde soll aus einer Art Katalog bestimmte Anforderungen auswählen, die das zu generierende System dann durch das Kombinieren von Komponenten aus einer Bibliothek erfüllt.
- Es ist unerheblich, ob das Endprodukt ein Softwaresystem oder nur ein Zwischenprodukt ist, welches noch manuell integriert werden muss.

3. Einschub: Generatoren

Generatoren sind die Schlüssel-Technologie in der generativen Programmierung. Von ihrer Implementierung hängt die Qualität der generierten Produkte ab. Grundsätzlich müssen Generatoren nicht unbedingt Quellcode oder Software erzeugen. Generatoren haben im Allgemeinen die Aufgabe, eine Eingabe in Sprache A in eine Ausgabe in Sprache B zu überführen. Eine Sprache kann hier jede beliebige Form zum Darstellen von Sachverhalten sein (UML, Text, Quellcode, Maschinencode etc.). So definiert, gehören auch Compiler, CASE-Tools, GUI-Builder oder Dokumentationswerkzeuge wie JavaDoc zu der Gattung der Generatoren.

Im Bezug auf das generative Programmieren betrachtet man jedoch ausschließlich jene Generatoren, die Softwarekomponenten als Ausgabe haben.

3.1. Definition: Software-Generator

A generator is a program that takes a higher-level specification of a piece of software and produces its implmentation. The piece of software could be a large software system, a component, a class, a procedure, and so on. [2]

3.2. Arbeitsweise von Software-Generatoren

Generatoren arbeiten nach dem EVA-Prinzip, d.h. es gilt die drei Arbeitsschritte - Eingabe, Verarbeitung und Ausgabe - sequenziell abzuarbeiten. Die Eingabe muss in einer für den Generator verständlichen Form vorliegen. Nachdem diese Eingabe validiert wurde, werden die noch offenen Aspekte, z.B. Abhängigkeiten zu Bibliotheken oder Unverträglichkeiten mit anderen Komponenten, mit Hilfe des Konfigurationswissens geklärt und die Eingabe damit vervollständigt. Die obligatorische Optimierung soll die effizienteste Kombination der dem Generator zur Verfügung stehenden Mittel finden und verbinden. Dann erfolgt die Generierung der Ausgabe.

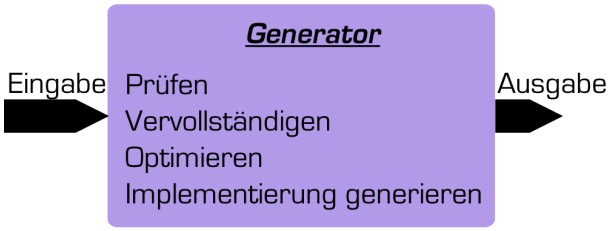

Abbildung 1. Arbeitsweise von SW-Generatoren.

3.3. Klassifikation

Die Überführung von einer Sprache in eine andere macht in der Softwareentwicklung nur dann Sinn, wenn sich entweder die Struktur oder aber die Abstraktion durch die

Transformation ändert. Deshalb lassen sich Generatoren in zwei Transformationsklassen einteilen, aus denen wiederum drei Arten von Generatoren resultieren.

- **Horizontale Transformation**
  Generatoren dieser Klasse verändern die Struktur des Programms, aber nicht die Abstraktionsebene. In diese Kategorie fallen z.B. Generatoren für die Aspekt-Orientierte Programmierung, bei der Module optimiert, ergänzt oder miteinander verwebt werden. Man spricht bei dieser Klasse auch von *Transformational Generators*

- **Vertikale Transformation**
  Generatoren dieser Klasse lassen die Struktur des Programms unverändert, modifizieren aber die Abstraktion. In diese Gattung fallen z.B. einfache Compiler, die die Hierarchie des Programms aufschlüsseln und in eine Low-Level-Sprache überführen. Auch CASE-Tools konkretisieren die abstrakten UML-Beschreibungen zu Source-Code und finden ihren Platz ebenfalls in dieser Kategorie. Man spricht hier auch von *Compositional Generators*. Die Transformation selbst nennt man auch *Forward Refinement*.

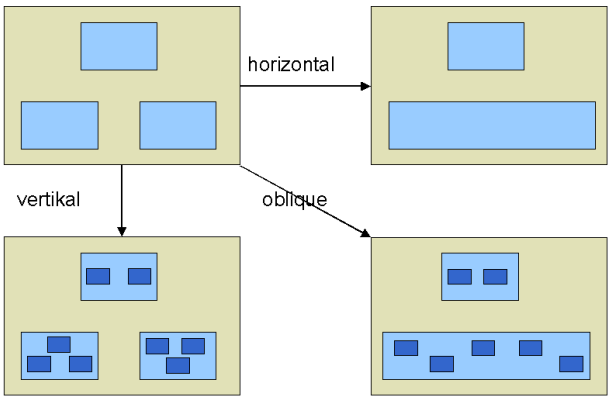

Abbildung 2. Transformationsklassen.

Diese beiden Transformationsklassen und deren Generatoren werden durch einen weiteren Generatortyp ergänzt: **Oblique Generatoren** gehören beiden Transformationsklassen an. Als Beispiel seien hier optimierende Compiler genannt, die neben dem Forward Refinement z.B. durch Inlining die Strukturgrenzen der Programme verändern.

3.4. Einsatzgebiete für Software-Generatoren

Möchte man flexible Systeme oder Produktlinien erstellen, kann jedoch aus Performancegründen nicht auf traditionelle generische OO-Konzepte zurückgreifen, macht der Einsatz von Software-Generatoren Sinn.

Diese Situation tritt gerade im Embeddedbereich häufig auf. Hier gibt es noch weitere Gründe für den Generatoreinsatz:

- Man möchte die Größe des Programmimages so klein wie möglich halten. Generierter Code wird nur die wirklich benötigten Teile enthalten.

- Man möchte den Ressourcenverbrauch oder die Schedulability analysieren. Dazu muss man den Code einer statischen Analyse unterziehen. Generierter Code kann einfacher analysiert werden als generischer.

Weiter kann man mit Generatoren die Wartbarkeit von Software deutlich erhöhen. Dafür eignet sich vor allem die Aspektorientierte Programmierung (AOP): Um eine Datenbankanwendung robuster zu gestalten möchte man vor jeder Datenbank-Abfrage sicherstellen, dass eine Verbindung zum Datenbank-Server hergestellt ist. Diese Abfrage müsste man ohne Generator in jede Funktion die auf die Datenbank zugreift implementieren. Darunter würde die Wartbarkeit enorm leiden. Ändert sich die Datenbankschnittstelle müssten all diese Stellen gefunden und angepasst werden. Mit AOP definiert man stattdessen, dass vor allen Datenbankzugriffen die Server-Verbindung überprüft werden soll. Der Generator webt diese Aspekte dann in den Code ein. Die Wartung findet dann nur noch an einer zentralen Stelle statt.

Besonders verbreitet ist das Generic Programming. Verwirrend ist hier die Namensgebung, denn es handelt sich zumindest bei dessen bekanntesten Vertreter durchaus um generatives Programmieren. Die Standard Template Library (STL) von C++ bietet typunabhängige Algorithmen und Container-Klassen wie Stacks, Listen, Maps etc. die zur Übersetzungszeit an einen Typ gebunden werden. In dem Kompilat findet man nun nicht wie bei den generischen Ansätzen eine Liste die alle Datentypen verwalten kann, sondern eine Liste die tatsächlich auf diesen einen gebundenen Typ spezialisiert ist.

Es ist möglich, dass sich manche Dinge in einer Programmiersprachen nicht ausdrücken lassen. Zum Beispiel kann man in Java im Rahmen eines Downcasts die Zielklasse nicht als Variable angeben. Generierter Code kann solche Probleme natürlich umschiffen. [4]

3.5. Beispiele für Software-Generatoren

Oben wurden kurze Beispiele für allgemeine Generatoren angegeben. Es folgen daher hier vier erfolgreiche Einsatzbeispiele für Software-Generatoren:

- **Java Server Pages**
  Java Server Pages ergänzen statische HTML-Seiten um dynamische Konstrukte, mit denen man z.B. Daten aus Datenbankabfragen in die HTML-Seite integrieren

kann. Diese Serverpages werden bei ihrem ersten Abruf vom Webserver in Java-Code transformiert, kompiliert und dann dort in einem Container ausgeführt.

- **Generative Matrix Computation Library (GMCL)**
  Die GMCL ist eine in C++ implementierte Matrixbibliothek, die für mehr als 1840 unterschiedliche Matrizentypen hochperformante Algorithmen zur Optimierung von Matrix-Ausdrücken bereitstellt. Der Benutzer kann für seine Datentypen konfigurieren, ob die Matrix z.B. dicht oder licht besetzt ist oder ob die Speicherzuweisungen statisch oder dynamisch erfolgen sollen. Der Generator/Präprozessor generiert aus diesen Konfigurationsparametern dann eine für diese Problemstellung optimale Algorithmenbibliothek.

- **Generative Matrix Factorization Library GMFL**
  Die GMFL erweitert die GMCL um effiziente LR-Zerlegungen verschiedener Matrizentypen.

- **Bordcomputer-Software für Satelliten**
  In der Satelliten-Steuerung sind ADA83-Programme im Einsatz, die aus einer XML-Spezifikation und mit Hilfe einer Templatesprache erzeugt wurden. Der Generator hat also ADA-Schnippsel zu einem angepassten Programm zusammengefügt.

4. Generatives Programmieren

Die generative Programmierung nutzt die Generatoren, um aus einer, in einer speziellen Sprache aufgeschriebenen Anforderungsspezifikation automatisch ein Software-Produkt zu erstellen welches diesen Anforderungen entspricht. Der Programmierer selbst soll dazu aber nicht in den Design- und Implementierungsprozess eingreifen.

Die speziell entwickelten Sprachen zur Spezifikation der Anforderungen nennt man *Domain Specific Language*, kurz *DSL*.

Es ist jedoch einiges an Vorarbeit zu leisten, bevor die Software auf Knopfdruck erstellt werden kann.

4.1. Prozess der Generativen Programmierung

Generative Programmierung wird in zwei Schritten angewendet.

Domain Engineering

"Domain Engineering is the activity of collection, organizing and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means of reusing these assets when building new systems."[2]

Das Domain Engineering ist die Vorarbeit der eigentlichen generativen Programmierung. Es umfasst Domain Analysis, Domain Design und Domain Implementation. Aus diesen drei Schritten resultiert ein Domänenmodell mit der DSL sowie der Generator selbst.

Application Engineering

Nach einem erfolgreichen Domain Engineering geht man zum Application Engineering über. Erst in diesem Schritt werden die eigentlichen Produkte erstellt. Der Kunde gibt in DSL eine Spezifikation an und der Generator erzeugt daraus das fertige Produkt. Eine solche Spezifikation könnte z.B. eine Featurematrix sein, bei der man bei allen gewünschten Features ein Kreuz macht.

4.2. Generative Domain Model

Der generativen Programmierung liegt das sog. Generative Domain Model zu Grunde. Es besteht aus drei Teilen: dem Problemraum, dem Lösungsraum und dem Konfigurationswissen, mit dessen Hilfe eine Spezifikation aus dem Problemraum in ein Produkt transformiert wird.

Problemraum

Für die Produktlinienentwicklung ist es wichtig, zu analysieren, in welchem Fachbereich die Produkte angewendet werden sollen und diesen genau zu untersuchen. Man beschränkt sich auf eine Teilwelt, deren derzeitige Anforderungen man genauso wie künftige erfasst. Man sammelt alle erdenklichen Features dieser Domäne (Domain Scoping) und identifiziert deren Abhängigkeiten und Unverträglichkeiten untereinander (Feature Modelling).

Aus diesem Wissen, erstellt man eine problemorientierte Beschreibungsprache. Diese *Domain Specific Language* gilt nur für die Domäne, die die Produktlinie bedienen soll. Das Ziel dieser Sprache ist es, die Anforderungen an ein spezielles Produkt aus der Produktlinie möglichst einfach und präzise formulieren zu können. Es ist unerheblich, ob diese Sprache textuell oder grafisch ist. Die in der DSL formulierte Produktspezifikation wird von dem Generator als Eingabe verstanden und bildet den Problemraum im Generative Domain Model.

Lösungsraum

Dem Problemraum gegenüber steht die Potenzmenge aller Komponenten der Produktlinie. Der Generator wählt gemäß der Spezifikation und mit Hilfe des Konfigurationswissens diese Komponenten aus und erzeugt das Produkt. Da Wiederverwendbarkeit in der GP im Vordergrund steht, muss man bei dem Entwurf der Komponenten auf die maximale Kombinierbarkeit und eine minimale Redundanz achten - so erreicht man eine große Anzahl von möglichen Produktvarianten.

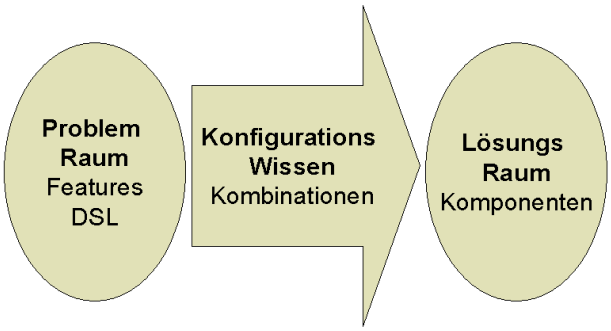

Abbildung 3. Generative Domain Model.

Konfigurationswissen

Das Konfigurationswissen ist in gewisser Weise der Bauplan der Produktlinie. Hier wird hinterlegt, welches Feature aus dem Problemraum in welcher Komponente des Lösungsraumes implementiert ist. Außerdem kann es vorkommen, dass sich manche Komponenten nicht kombinieren lassen. Damit der Generator das nicht trotzdem tut, muss diese Information hier hinterlegt werden.

5. Vorteile und Nachteile

5.1. Vorteile

Generative Programmierung klingt verlockend. Nicht nur die Analyse sondern auch die Implementierung der einzelnen Komponenten wird aus dem eigentlichen Produktentwicklungsprozess herausgezogen. Nach einem gründlichen Domain Engineering führt dies zu wesentlich kürzeren Analyse- und Entwicklungszeiten für die einzelnen Produktvarianten. Durch die Verwendung einer speziellen Spezifikationsprache, kann man die Anforderungen direkt in ein Produkt überführen, ohne dass ein Entwickler die Spezifikation zuvor analysieren, verstehen und umsetzen muss. All das macht der Generator.

Im Bezug auf Performance und Ressourcenverbrauch werden die Produkte vom Generator optimiert. Generische Konzepte können das nicht leisten. Gegenüber diesen muss der Entwickler auch deutlich weniger Code schreiben, dadurch fallen weniger Fehler in dem Produkt an. Findet sich doch ein Fehler, so wird er zentral in der betroffenen Komponente im Lösungsraum behoben. Nach dem nächsten Generatorlauf wird er in allen Produkten nicht mehr vorkommen.

5.2. Nachteile

Auf der anderen Seite schränkt die GP die Entwicklungsmöglichkeiten auch ein. Das kann besonders schwerwiegend sein, wenn Features im Domain Engineering übersehen wurden. Es ist nämlich nicht möglich, den generierten Code manuell anzupassen. Um vergessene Features nachträglich hinzuzufügen muss man das aufwendige Domain Engineering wiederholen.

Der Weg zu einem funktionierenden generativen Entwicklungsprozess ist außerdem sehr lang. Nach der Analyse muss die komplexe Komponentenbibliothek erstellt werden. Daher kann es Jahre dauern bis das erste Produkt aus der Produktlinie generiert werden kann. Das macht nur dann Sinn, wenn das Entwicklungsteam bereits über viel Erfahrung und Wissen über die Domäne verfügt und sie gut abgrenzen kann.

6. Fazit

Da Software hauptsächlich zur Automatisierung von Prozessen genutzt wird, kann man sich berechtigter Weise fragen, warum die komplexen Prozesse der Softwareentwicklung selbst kaum automatisiert werden. Generative Programmierung stellt hier einen interessanten Ansatz in der Produktlinienentwicklung dar. Die Autoren des wohl wichtigsten Werks über Generatives Programmieren [2] lassen durchklingen, dass in näherer Zukunft die konventionelle Softwareentwicklung durch die generative ersetzt werden wird. Allerdings muss man auch sehen, wann der Einsatz von Generatoren überhaupt Sinn macht. Für ein einzelnes Produkt macht sich die Analyse und die Entwicklung des Generators nicht bezahlt. Und auch bei Produktlinien lässt sich die Komplexität nicht komplett durch einen Generator abdecken. Ohne die manuelle Softwareentwicklung wird es also so bald nicht gehen - dennoch kann die generative Programmierung die manuelle sehr wohl ergänzen und damit vereinfachen.

Literatur

[1] K. Czarnecki und U. W. Eisenecker. Components and generative programming (invited paper). Aus *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Seiten 2–19. Springer-Verlag, 1999.

[2] K. Czarnecki und U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[3] U. W. Eisenecker. Von der Einzelanwendung zur Systemfamilie, 2002. Folien, 28 Seiten, http://www.saxos.ch/meta/eisenecker.pdf.

[4] M. Völter. Ghost Writer - Wann ist Code-Generierung sinnvoll? *Java Magazin*, (10/03), 2003.