Seminar „Moderne Softwareentwicklung“ SS2006

Einführung in Generatives Programmieren

Bastian Molkenthin

Betreuer: Volker Kuttruff

21/ Juli 2006

Zusammenfassung

Während die Industrie ihre Produkte am Fließband von Robotern vollautomatisch zusammensetzt, hinkt die heutige Softwareentwicklung dieser Produktivität weit hinterher. Generatives Programmieren soll diese Lücke verkleinern, indem der Übergang von Einzelprodukten hin zur Herstellung ganzer Softwarefamilien vollzogen wird. Mit dem gewonnenen Wissen aus dem Domain Engineering wird eine spezielle Domänensprache erstellt, mit welcher das zu erzeugende Produkt beschrieben wird. Diese Spezifikation wird von einem Generator eingelesen, welcher auf Knopfdruck verschiedene hochoptimierte Softwarevarianten automatisch generieren kann. Dieses Dokument soll einen Einblick in die Ideen, Methoden und Werkzeuge des generativen Programmieren geben und abschließend die wichtigsten Vor- und Nachteile darstellen.

1 Einführung

1.1 Einleitung

Zu Beginn der Softwareentwicklung war die Komplexität von Programmen leicht überschaubar. Mit der immensen Verbreitung und der stetigen Leistungssteigerung von Computern sind auch die Anforderungen an Qualität, Produktivität, Flexibilität und Komplexität gewachsen. Diese Entwicklung lässt sich mit der Industrialisierung vergleichen. 1826 gelang es John Hall, Musketen komplett aus austauschbaren Teile herzustellen – analog zur modularen Softwareentwicklung, deren Komponenten auch in anderen Systemen wieder verwendet werden. Trotzdem blieb die Herstellung kostspielig und zeitaufwendig, bis 1901 das Fließband eingeführt und 1913 durch Henry Ford verbreitet wurde. Im Bereich der Softwareentwicklung kann dieser Fortschritt mit der Produktlinien-Architektur und der Objektorientierung verglichen werden. Doch während die Softwareentwicklung an diesem Punkt stehen geblieben ist, ist die produzierende Industrie einen Schritt weiter: Industrieroboter fertigen am Fließband vollautomatisch verschiedene Varianten eines Produkts. Generatives Programmieren soll diesen Rückstand durch Automatisierung von Produktionsschritten aufholen.

1.2 Rückblick auf OO-Entwurf

In den neunziger Jahren hat sich das Prinzip der objektorientierten Softwareentwicklung in vielen Bereichen durchgesetzt. Viele größere Projekte werden heutzutage objektorientiert modelliert, spezifiziert und implementiert. Unterstützt wird dieser Trend durch neue Programmiersprachen wie beispielsweise Java und C# als auch durch Modellierungssprachen wie UML und moderne Entwurfsmuster, welche allesamt auf das Prinzip der Objektorientierung zugeschnitten sind.. Diese Entwicklung ist auf viele nützliche Neuerungen gegenüber der bis dahin üblichen imperativ-prozeduralen Sprachen zurückzuführen. Folgende Auflistung fasst die wichtigsten Vorteile zusammen.

- Klassen und Objekte: Erlauben die Beschreibung gemeinsamer Strukturen und gemeinsamen Verhaltens eines Objekts mithilfe von Attributen und Methoden (Klassifizierung). Ziel ist die Abstraktion und Modellierung eines Systems.
- Vererbung: Ermöglicht die Übernahme von Funktionalität von einer Klasse zu einer anderen und stellt somit ähnliche Klassen in Beziehung bzw. baut eine Klassenhierarchie auf.
- Polymorphie und dynamisches Binden: Erlaubt das Schreiben von Code für verschiedene Typen und das Variieren dieser Typen zur Laufzeit.
- Kapselung und Information Hiding: Die interne Struktur von Objekten wird nicht offen gelegt und bietet Schutz sowie Modularisierung einzelner Komponenten. Somit wird der direkte Zugriff auf die interne Datenstruktur unterbunden, der Zugriff kann nur über eine öffentlich definierte Schnittstelle erfolgen.

1.3 Nichtgelöste Probleme

Im Laufe der Zeit hat sich jedoch in der Realität gezeigt, dass nicht alle wünschenswerten Aspekte mit Hilfe des objektorientierten Entwurfs erfüllt werden. Besonders in den folgenden Gebieten sind nach wie vor Defizite zu verbuchen:

- Wiederverwendbarkeit: Obwohl Klassen und Frameworks für diesen Zweck entwickelt wurden, zeigen sich in der Realität immer wieder Probleme mit der Wiederverwendbarkeit. Klassen sind als Wiederverwendungseinheit zu klein, Frameworks hingegen schwer zu integrieren. Zudem verläuft die Entwicklung oft ad-hoc anstatt systematisch, welches die spätere Verwendung in anderen Gebieten erschwert oder sogar nicht ermöglicht: wichtige Aspekte werden übersehen oder nur unzureichend beachtet, das Design ist schlicht zu inflexibel. Es fehlen einfach bessere Methoden zur Entwicklung wiederverwendbarer Software.
- Handhabung der Komplexität: Nachträgliche Änderungen implizieren zumeist neue (abgeleitete) Klassen oder die Erweiterung vorhandener Klassen, was zur Fragmentierung des Designs und wachsender Komplexität führt. Aber auch das ‚Einflicken' von Features wie Synchronisation oder Optimierung tragen allmählich zu diesem Trend bei. Zusätzlich ist zu erwähnen, dass Entwickler oft zu einer 'Übergeneralisierung' auf Kosten der Speicheressourcen und der Laufzeitperformance tendieren.
- Verlust von Informationen: Trotz Definitionen, Spezifikationen und Kommentaren gehen Informationen auf dem langen Weg von der Anforderungsanalyse bis hin zum geschrieben Quellcode verloren. Die semantische Lücke zwischen Abstraktion der Realität und Features von Programmiersprachen ist groß. Dieses Phänomen ist oft bei der Instandhaltung, Weiterentwicklung und Portierung von Software auf andere Plattformen zu beobachten: mit zunehmendem Alter muss mehr Zeit und somit mehr Kosten hineingesteckt werden, sodass veraltete Software meistens früher oder später durch komplett neugeschriebene Programme ersetzt werden.

Die hier aufgelisteten Probleme treten nicht nur in OO Methoden, sondern in nahezu allen Entwicklungsstrategien auf. Da eine Verbesserung nicht in Sicht ist, wird nach neuen Methoden und Strategien in der Softwareentwicklung gesucht.

2 Generatives Programmieren

2.1 Ziele des Generativen Programmierens

Hauptziel des generativen Programmieren (GP) ist eine weit bessere Wiederverwendbarkeit von Softwarekomponenten. Die bisherige Ausrichtung auf Einzelsystementwicklung soll zu Gunsten von Systemfamilienentwicklung verlassen werden. Es soll mit geringem Aufwand möglich sein, mehrere Varianten eines Produkts herzustellen, also eine schnellere Anpassung an verschiedene Problemstellungen vorzunehmen. Dies soll mit Hilfe von so genannten Generatoren geschehen, indem nach Analyse der Domäne und Spezifizierung des Problems in

einer domänenspezifischen Sprache ein Generator die Softwarekomponenten erzeugt. Somit wird dem Entwickler die Aufgabe abgenommen, per Hand die Umsetzung des anwendungsspezifischen Problems im Lösungsraum zu realisieren. Zum einen verspricht man sich aufgrund automatischer Quellcodegenierung weniger Fehler im fertigen Produkt, zum anderen Optimierung im Hinblick auf Speicher, Geschwindigkeit und weiteren Eigenschaften, da das System äußerst gut auf die Problemlösung abgestimmt ist.
Doch auch wirtschaftliche Vorteile sollen durch die Automatisierung des Software-Entwurfs erreicht werden. Die dadurch entstehende verkürzte Entwicklungszeit impliziert auch eine höhere Produktivität, ähnlich der Fließbandverarbeitung in der Industrie. Der Entwickler kann somit in kürzeren Zeitabständen qualitativ höherwertige Software liefern und somit höhere Gewinne verbuchen. Der Kunde hingegen profitiert von früher fertig gestellter Software bei gleichzeitig niedrigeren Kosten.
Das GP beschleunigt und verbessert aufgrund von Automatisierung und Wiederverwendbarkeit den kompletten Prozess der Softwareentwicklung.

2.2 Erklärung wichtiger Begriffe

2.2.1 Domäne

Je nach Bereich unterscheidet sich die Bedeutung des Begriffs „Domäne“ beträchtlich, trotzdem lassen sich zwei generelle Gebiete ausmachen.

- Domäne der realen Welt: beinhaltet das Wissen über Konzepte und Abläufe eines bestimmten Problembereichs.
- Domäne als Menge von Systemen: umfasst das Wissen der realen Welt als auch das Wissen zur Erstellung von Softwaresystemen.

Jede weitere Benutzung des Begriffs „Domäne“ in diesem Wort bezieht sich auf letztere Definition.

2.2.2 Generatives Domänenmodell

Generatives Programmieren richtet den Blick auf die Automatisierung der Herstellung von Softwareprodukten, von Komponenten bis hin zu kompletten Applikationen. Dabei werden die einzelnen Mitglieder einer Softwarefamilie auf Basis eines gemeinsamen generativen Domänenmodells erstellt; einem Modell einer Systemfamilie, welches aus drei Teilen besteht: dem Problemraum, dem Lösungsraum und dem Konfigurationswissen.
Der Problemraum besteht aus Konzepten, Merkmalen und Begriffen, die ein Anwendungsprogrammierer, vor allem aber der Kunde, benutzt, um ein spezielles Produkt herzustellen. Hierzu wird eine problemorientierte, spezialisierte Sprache benutzt, domänenspezifische Sprache (DSL) genannt.
Der Lösungsraum setzt sich hingegen aus den Komponenten mit all ihren Kombinationen zusammen, er definiert sozusagen alle möglichen Varianten einer Softwarefamilie und stellt damit die technische Realisierung dar. Die Implementierung der Komponenten sollte im Hinblick auf maximale Kombinierbarkeit erfolgen, d.h die Komponenten sollten in möglichst

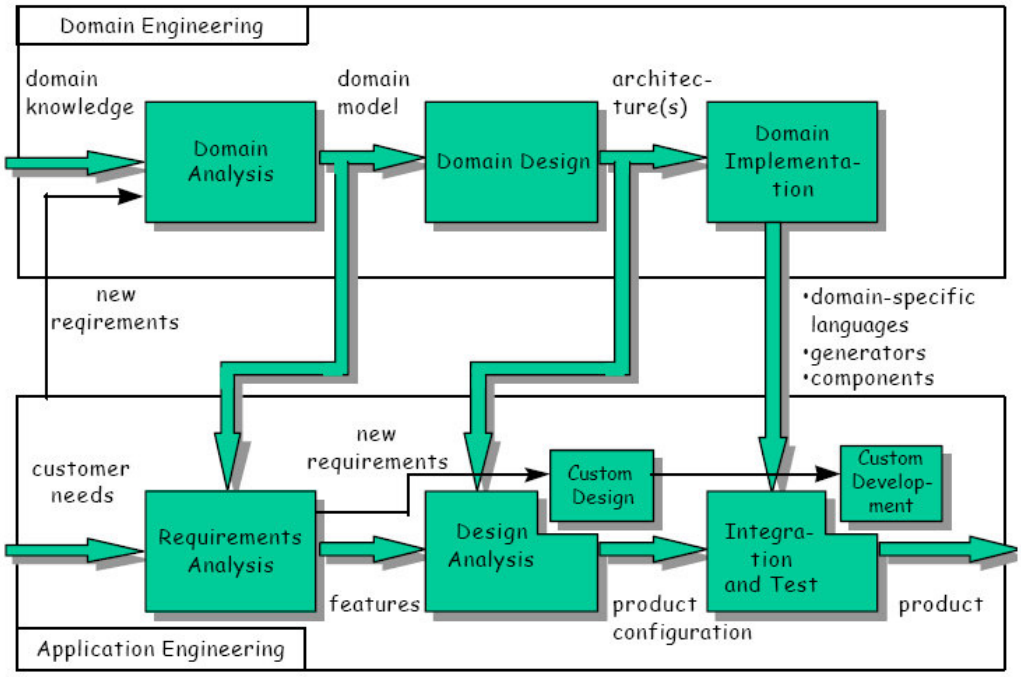
vielen Weisen zusammen benutzt werden können; aber auch im Hinblick auf minimale Redundanz zur Vermeidung von Codewiederholung und maximale Wiederverwendbarkeit.
Das Konfigurationswissen schließlich ist das Verbindungsstück zwischen Problem- und Lösungsraum. Hier werden beispielsweise nicht mögliche Kombinationen von Komponenten, nötige Abhängigkeiten, die aus den Merkmalen resultieren, und Konstruktionsregeln spezifiziert. Auch Optimierung spielt eine Rolle, um die bestmögliche Zusammenarbeit der Komponenten zu gewährleisten. Im Falle einer ungenügenden Angabe von wichtigen Aspekten beinhaltet das Konfigurationswissen Standardeinstellungen.
Das generative Domänenmodell stellt somit das Prinzip des generativen Programmierens dar: nach Spezifierung im Problemraum erstellt der Generator, der das Konfigurationswissen enthält, ein Produkt, welches einer Komponentenkombination im Lösungsraum entspricht.

2.2.3 Domänenspezifische Sprache

Eine domänenspezifische Sprache, abgekürzt DSL (aus dem Englischen von 'Domain-Specfic Language'), ist eine problemorientierte Sprache, die auf eine bestimmte Domäne zugeschnitten ist. Sie spielt eine zentrale Rolle im generativen Programmieren, da mit ihr die Spezifikation der zu erstellenden Software beschrieben wird. Beim Entwurf der DSL muss darauf geachtet werden, dass alle möglichen Spezifikationen der Domäne erfasst werden können, da sie im Grunde das einzige Werkzeug des Entwicklers sind, um ein Softwareprodukt zu erstellen. Ziel ist es, nur durch Änderungen in der DSL das Zielprodukt variieren zu können, doch in der Realität wird bei kleinen Modifikationen häufig auch nur das Resultat berichtigt. Zudem sollte auf eine gewisse Abstraktion Wert gelegt werden: sie sollte Begriffe, Merkmale und Denkweise eines Domänenexperten widerspiegeln und keinesfalls Annahmen über die Zielmaschine oder Ähnlichem enthalten. Die fertige Spezifikation wird anschließend von einem Generator eingelesen, welcher das Zielprodukt erstellt. Somit können in der DSL auch ungültige Beschreibungen existieren, da der Generator diese erkennen kann. Die Form der DSL ist frei wählbar: sie kann textuell (SQL, EBNF, ...) oder graphisch (ähnlich wie in einem GUI-Editor) sein.

3 Methoden des Generativen Programmierens

Ziel des GP ist es, den Wandel von der Produktion von Einzelsystem hin zu der Produktion von Systemfamilien zu vollziehen. Hierzu müssen zunächst die nötigen Informationen wie Komponenten und Richtlinien gesammelt werden, was im so genannte Domain Engineering geschieht. Anschließend wird im darauf folgenden Schritt namens Application Engineering ein konkretes Produkt aus passenden Komponenten zusammengesetzt. Das GP soll hierbei mögliche Vorgehensweisen anbieten.



3.1 Domain Engineering

In [CE00] definieren Czarnecki und Eisenecker das Domain Engineering wie folgt: „Domain Engineering ist das Sammeln, Organisieren and Speichern bisheriger Erfahrungen im Erstellen von System oder Teilen von Systemen in bestimmten Domänen in der Form von wieder verwendbaren Komponenten, sowie das zur Verfügungstellen von angemessenen Mitteln, um diese Komponenten beim Erstellen neuer Systeme wiederzuverwenden“.
Das Domain Engineering ist somit die komplexe und zeitaufwendige Vorarbeit des generativen Programmierens, welche die Wiederverwendbarkeit in der Softwareentwicklung erst ermöglicht („development for reuse“). Es lässt sich auf eine Vielzahl von Problemen anwenden, wie die Entwicklung domänenspezifischer Frameworks und Sprachen, Komponentenbibliotheken und Generatoren. Domain Engineering setzt sich aus den drei Hauptkomponenten Domain Analysis, Domain Design und Domain Implementation zusammen.

- Domain Analysis: Aufgabe der Domain Analysis ist es, die Domäne abzugrenzen, die relevanten Informationen zu sammeln und schließlich in ein Domänenmodel zu integrieren, welches die Eigenschaften, Gemeinsamkeiten und Abhängigkeiten innerhalb der Domäne repräsentiert. Informationsquellen umfassen beispielsweise existierende Systeme in der Domäne, Expertenwissen oder aber auch Experimente
- Domain Design: Hier findet nun mit den gesammelten Daten der ersten Phase die Entwicklung einer allgemeinen Systemarchitektur für die komplette Systemfamilie statt.

- Domain Implementation: In der letzten Phase erfolgt nun die Erstellung der Architektur, der Komponenten und eines Generators, welche das Fundament für das anschließende Application Engineering bilden.

3.1.1 Feature Modeling

Feature Modeling beschreibt den Prozess, allgemeine und variable Eigenschaften von Konzepten und ihren Abhängigkeiten zu modellieren und diese dann organisiert in einem so genannten Featuremodell darzustellen. Konzepte bedeuten in diesem Zusammenhang alle Elemente und Strukturen in einer Domäne.
Somit stellt Feature Modeling den größten Beitrag zum Domain Engineering dar, da diese Technik der Schlüssel zum Finden und Einordnen variabler Elemente beschreibt, welche verständlicherweise in weit größerer Anzahl vorkommen als beim Entwurf eines konkreten Produkts. Somit hilft es dem Entwickler in zwei wichtigen Punkten. Zum Einen vermindert es die Wahrscheinlichkeit, dass relevante Features übersehen werden. Zum anderen, dass nicht benötigte Features eingeschlossen werden, die unnötigerweise die Komplexität und somit auch die Entwicklungskosten erhöhen. Das resultierende Featuremodell gibt eine präzise, abstrakte und explizite Repräsentation der vorkommenden Variabilität wieder.
Die bekannteste Methode ist das Feature-Oriented Domain Analysis (FODA), welches am Software Engineering Institute (SEI) in Pittsburgh entwickelt wurde.

3.2 Application Engineering

Application Engineering ist der Prozess der Erstellung kompletter Systeme, basierend auf den im Domain Engineering gewonnenen Erkenntnisse. Aus einer Spezifikation wird unter Verwendung des Domänenmodells und des Generators ein konkretes Produkt, z.B eine Softwarevariante einer Produktfamilie, generiert.
Während der Anforderungsanalyse werden die vom Kunden gewünschten Features einer bestimmten Applikation in das bereits existierende Domänenmodell eingefügt. Das Ergebnis dieses Schrittes ist eine Spezifikation des herzustellenden Produkts in einer DSL, welches an die automatische Systemherstellung, den Generator, übergeben wird. Dieser nimmt nun die eigentliche Implementierung vor, indem er die zur Verfügung stehenden Komponenten so kombiniert, dass als Endprodukt eine optimale Softwarelösung resultiert.
Der Schritt des Application Enginnering kann mehrfach wiederholt werden, um beliebig viele verschiedene Versionen einer Software zu genieren. Dabei werden jedes Mal die Ergebnisse des Domain Engineering verwendet, weswegen das Application Engineering auch als „development with reuse“ umschrieben wird.
Falls Anforderungen eines Kunden noch nicht im Domänenmodell vorhanden sind, müssen diese manuell entwickelt werden und sollten anschließend dem Domain Engineering übergeben werden, damit sie später als wiederverwendbare Komponenten benutzt werden können. Anschließend kann das Produkt automatisch unter Verwendung des Generators aus den neuen Anforderungen und den bisher existierenden Komponenten zusammengesetzt werden. Das DE und das AE sind somit nicht zwei hintereinander ablaufende, getrennte Prozessphasen, sondern können unter Umständen auch Hand in Hand gehen.

4 Generatoren

4.1 Definition / Beschreibung

Generatoren implementieren das generative Domänenmodell. Sie beinhalten das Konfigurationswissen und übernehmen die Zusammenstellung der Komponenten gemäß der Systemspezifikation. Laut Definition ist ein Generator ein Programm, das aus einer Spezifikation einer Softwarekomponente (meist in einer DSL verfasst) dessen Implementierung herstellt.
Im Allgemeinen führt ein Generator folgende vier Aufgaben durch:

- Überprüfen der Eingabe: Falls die Spezifikation in Textform vorliegt, wird die Eingabe mit Hilfe eines Lexers und eines Parsers in eine interne Repräsentation umgewandelt, ähnlich eines Compilers. In seltenen Fällen kann die Eingabe auch als Datenstruktur vorliegen, die beispielsweise mit einem speziellen Editor erstellt wurde. Selbstverständlich werden auch Warnungen und Fehler, falls nötig, ausgegeben.
- Vervollständigung der Eingabe: Lücken in der Spezifikation werden mit Standardeinstellungen gefüllt. Dies erlaubt es dem Benutzer, nur relevante Daten anzugeben und erst bei Bedarf genauer zu spezifizieren.
- Optimierung
- Generierung der Implementierung

Je komplexer die Spezifikationen werden, desto mehr wächst auch die Komplexität der Generatoren an, weswegen ihr interner Aufbau meistens modularisiert wird: somit besteht ein Generator aus einer Menge von kleineren, zusammenarbeitenden Generatoren.

4.2 Beispiele für Generatoren

Schon heutzutage kommen Generatoren verschiedenster Arten zum Einsatz. Eingabedaten in einer speziellen Notation werden, falls korrekt, von einem Generator in Ausgaben anderer Form umgewandelt.
Das Standardbeispiel sind Compiler: ein Quelltext wird von einem Lexer und einem Parser in eine interne Datenstruktur eingelesen und gleichzeitig auf Korrektheit überprüft. Im Falle von Fehlern werden entsprechende Meldungen zurückgegeben. Lücken in der Eingabe werden teilweise vervollständigt, zum Beispiel werden nichtinitialisierten Variablen Standardwerte zugewiesen. Zuletzt wird der Maschinecode für die jeweilige Plattform optimiert und generiert.
Ebenso gehören hier auch moderne Entwicklungsumgebungen hinzu, die in der Lage sind, aus UML Diagrammen automatisch Javacode zu erzeugen.
Diese Generatoren sind aber doch relativ simpel und entsprechen nicht ganz den Vorstellungen, wie sie in [CE00] formuliert werden, weswegen hier weitere (unbekanntere) Beispiele folgen, welcher der Idee des GP besser nachkommen:

- Die „Generative Matrix Computation Library“ (GMCL) ist ein Musterbeispiel für GP. Durch Verwendung verschiedener Programmiertechniken unterstützt die entworfene Matrix-Konfigurations-DSL mehr als 1840 unterschiedliche Arten von Matrizen, wobei die C++ Implementierung nur ca. 7500 Codezeilen umfasst.
- Die „Generative Matrix Factorization Library“ (GMFL) generiert Algorithmen für LU-Zerlegungen von Matrizen, basierend auf der GMFL, bei nur rund 3800 Zeilen Code.
- Ein weiteres Beispiel ist die Bordcomputer-Software für Satelliten. Die Spezifikation erfolgt in XML, aus der mit Hilfe einer JSP-ähnlichen Templatesprache Ada83-Code erzeugt wird.
- Auch die Statistikbibliothek für die Postautomatisierung der Firma Siemens Electrocom sollte hier erwähnt werden. Diese generiert verschiedene Arten von Zählern, Zeitgebern und statistischen Algorithmen.

5 Programmiertechniken für Generatives Programmieren

Um das Prinzip des GP in die Tat umzusetzen, werden neue Programmiertechniken für die Implementierung der Generatoren benötigt. Ziel für die erzeugten Komponenten sind Orthogonalität, minimale Codewiederholung und gleichzeitig hohe Kombinierbarkeit, um eine größtmögliche Wiederverwendbarkeit zu erreichen. Die wichtigsten Prinzipien hierzu sollen kurz erläutert werden.

5.1 Generisches Programmieren

Generisches Programmieren, bekannt geworden durch die Standard Template Library (STL) in C++, umfasst die abstrakte, generalisierte Beschreibung von Klassen, Funktionen, Datenstrukturen und Algorithmen zur Maximierung von Anpassungsfähigkeit und Kompatibilität. Dies erlaubt die Verwendung von Komponenten mit unterschiedlichen Datentypen. Bei Implementierung einer konkreten Software wird automatisch die passenste Instanz gewählt, ohne das die Effizienz beeinträchtigt wird. Im Kontext des GP repräsentiert das generische Programmieren eine wichtige Technik, um den Lösungsraum im generativen Domänenmodell zu organisieren, also die Komponenten, aus denen das Produkt zusammengesetzt wird. Wichtige Konzepte sind Typ-Parametrisierung und Polymorphie. Generisches Programmieren wird in Programmiersprachen wie C++ (realisiert durch Templates), Java ab Version 1.5, Ada, Eiffel und den .NET Sprachen unterstützt.
Generisches und generatives Programmieren werden häufig verwechselt bzw. als Synonym verwendet. Hauptunterschied ist folgender: Der Fokus der generischen Programmierung liegt einzig und allein auf der Repräsentation von Domänenkonzepten, während das generative Programmieren auch das Bilden von konkreten Instanzen umfasst.

5.2 Metaprogrammierung

Als Metaprogrammierung beschreibt man die Entwicklung von Programmen, die ihrerseits andere Programme als Dateneingabe bekommen und diese generieren oder modifizieren. Das wohl bekannteste Beispiel hierfür sind Compiler, die Programme aus einer Hochsprache in ihre äquivalente Darstellung in Maschinensprache umwandeln.
Metaprogrammierung stellt ein fundamentales Instrument zur technischen Realisierung des Konfigurationswissens dar, dem Verbindungsstück zwischen Problem- und Lösungsraum, und erst sie ermöglichen die Entwicklung von mächtigen Generatoren.
Programmiersprachen, die Metaprogrammierung in ihrem Quellcode zulassen, sind beispielsweise C++ und Lisp. In C++ wird dies durch so genannte Templates realisiert, die schon zur Kompilierzeit ausgewertet werden und zu diesem Zeitpunkt den eigentlichen Code schreiben. Aufgrund der Turing-Vollständigkeit der Template-Metaprogrammierung in C++ lässt sich theoretisch jeder Algorithmus so darstellen.

5.3 Aspektorientiertes Programmieren (AOP)

Modularisierung ist ein fundamentales Programmier- und Entwicklungskonzept. Die Lokalität von Quellcode eines bestimmten Problems bringt viele Vorteile mit sich: zum Einen erleichtert es dem Entwickler, sich im Code zurechtzufinden. Zum anderen vereinfacht es die Analyse, Erweiterbarkeit, Wiederverwendbarkeit und Abänderung, da sich alles kompakt an einer Stelle befindet. Es gibt jedoch auch Sachverhalte, in denen es sehr schwierig oder gar unmöglich ist, das saubere Systemdesign zu erhalten, wie beispielsweise das Hinzufügen von Sicherheits- oder Synchronisationsunterstützung. Die Implementierung jedes weiteren solchen Features fragmentiert den Code oder bedeutet gar ein komplettes Redesign, um die Performance zu erhalten. AOP soll hier Abhilfe schaffen, indem es Methoden und Techniken zur Verfügung stellt, Probleme in funktionale Komponenten als auch in Aspekte aufzuteilen. Derzeitige Programmiersprachen erlauben es leider noch nicht befriedigend, Entscheidungen im Design und im Entwurf nach dem AOP-Prinzip zu trennen.

6 Vor- und Nachteile

6.1 Vorteile

Die Vorteile des GP sind klar ersichtlich. Die verkürzte Enwicklungsdauer von neuen Softwareprodukten verringert die Herstellungskosten, was sowohl dem Hersteller als auch dem Kunden eklatante Vorteile verschafft. Die bequeme Spezifizierung mit Hilfe der DSL nimmt dem Entwickler die Aufgabe ab, bei jeder neuen Produktvariante etwaige neue Anforderungen per Hand in Form von Datenstrukturen und Algorithmen zu programmieren. Das Abstraktionsniveau wird erhöht, da man nur benötigte Features und nicht konkrete Komponenten einer Bibliothek angeben muss. Stattdessen können Generatoren neue Versionen per Knopfdruck vollautomatisch generieren und sind zusätzlich in der Lage, ungültige Spezifikationen zurückzuweisen und Optimierungen am Produkt in Bezug auf

Ressourcenverbrauch und Performance vorzunehmen. Dies vermindert die Fehleranfälligkeit, da der Entwickler selbst weniger Code für das eigentliche Produkt schreiben muss. Da ein Generator in den meisten Fällen wiederum aus mehreren kleineren Generatoren zusammengesetzt ist, ist die Möglichkeit vorhanden, einzelne auch in komplett anderen Projekten erneut einzusetzen. Ebenso ist zu erwähnen, dass zusammen mit dem Domain Engineering und der DSL ein Großteil des erarbeiteten Designwissens in Programmform vorliegt anstelle von Dokumenten und Diagrammen und somit spätere Weiterentwicklung vereinfacht. Als letzten Punkt sollte noch angemerkt werden, dass man den Übergang zur generativen Programmierung langsam und iterativ angehen kann. Entwickler müssen nicht sofort nach dem höchsten Level der Automatisierung streben, erst recht nicht bei kleineren Softwareprojekten. Es ist durchaus möglich, nur der Projektgröße angemessene Aspekte des GP anzuwenden, wie beispielsweise Feature Modeling, ohne die Entwicklung eines Generators.

6.2 Nachteile

Auf der anderen Seite ergeben sich durch das GP auch neue Probleme. Die Entwicklung wiederverwendbarer Komponenten ist komplexer und nimmt mehr Zeit in Anspruch als eine Implementierung, die nur für die Lösung eines einzigen, speziellen Problems entworfen wurde. Generatoren selbst stellen somit eine sehr umfangreiche Software dar, deren Herstellung und Wartung hohe Kosten verursacht. Dieser Mehraufwand macht nur Sinn, wenn als Ziel die Vermarktung einer ganzen Produktfamilie verfolgt wird. Auch der Aufwand des Domain Engineering kann nicht vernachlässigt werden, da diese Phase das Fundament des GP darstellt. Unzureichende Analyse der Domäne kann dazu führen, dass wichtige Aspekte übersehen werden. Diese können folgerichtig nicht mit der DSL formuliert werden und somit die Erstellung des Produkts verhindern, da nachträgliche Codeänderungen im Modell des GP nicht vorgesehen sind. Dies kann ein erneutes Domain Engineering und die Erweiterung des Generators zur Folge haben. Des Weiteren sollte auch die Abgrenzung der Domäne mit äußerster Sorgfalt geschehen. Eine sehr breit gewählte Domäne kann nach einem zu allgemeinen, nicht mehr realisierbaren Generator verlangen, während eine zu kleine Domäne Einschränkungen für die Softwarefamilie bedeuten kann.

Literatur

[CE00] Krzysztof Czarnecki, Ulrich W. Eisenecker. *Generative Programming – Methods, Tools, and Applications.* Addison-Wesley Verlag, 2000

[C01] Krzysztof Czarnecki. *Generative Programmierung und die Entwicklung von Softwarefamilien.* DaimlerChrysler AG
http://www.tu-chemnitz.de/informatik/Betriebssysteme/IFKOLL/2001/docs/k53.pdf

[C02] Krzysztof Czarnecki. *Brief Overview of. Generative Programming.* DaimlerChrysler Research and Technology Ulm, Germany
http://www.cwi.nl/events/2002/GP2002/slides/czarnecki-gp-intro.pdf

[LES] Tammo van Lessen. *Generatives Programmieren.*
http://www.taval.de/pub/tl-folien.pdf

[VOE] Daniel Voelz. *Generative Programmierung.* Universität Leipzig
http://www.bis.uni-leipzig.de/download/sem_mmk/Daniel_voelz_pdf.pdf