



Prof. Dr.-Ing. Mathias Sporer

erwarb nach einer Berufsausbildung zum Facharbeiter für Datenverarbeitung an der TU Dresden im Fernstudium den akademischen Grad des Diplom-Informatikers. Thema der Diplomarbeit war die ereignisorientierte diskrete Simulation technischer Prozesse. Diese Aktivitäten wurden an der TU Chemnitz fortgesetzt und führten zur Promotion auf dem Gebiet der Informatik mit dem Schwerpunkt der Datenbankunterstützung für den Entwurfsprozess eingebetteter Systeme.

In der Industrie arbeitete Mathias Sporer auf dem Gebiet der System- und Anwendungsentwicklung für Mainframe- und Personal-Computer im Umfeld netzwerkorientierter und relationaler Datenbanksysteme.

Nach Lehrtätigkeiten an der TU Chemnitz und der Staatlichen Studienakademie Glauchau erfolgte im Jahre 2010 die Berufung zum hauptamtlichen Dozenten für Informatik und im Jahre 2017 zum Professor an der Berufsakademie Sachsen. Die Forschungsinteressen liegen auf dem Gebiet der datenbankgestützten Modellierung und Simulation.

Kontakt: mathias.sporer@ba-sachsen.de

Generative Programmierung mit Bordmitteln

Mathias Sporer

Generative Programmierung ist ein sprachübergreifendes Konzept zur dynamischen Erzeugung von Programmcode auf der Grundlage von Anwenderanforderungen. Der folgende Beitrag beschreibt Einsatzmöglichkeiten und Grenzen dieses Verfahrens.

Generative programming is a cross-language concept for dynamically generating program code based on user requirements.

Die Entwicklung von Computerprogrammen ist ein zeit- und kostenintensiver Prozess, da die Anforderungen einer Gruppe von Nutzern manuell in kleine, von Rechnersystemen ausführbare Schritte zerlegt werden müssen. Deshalb gab es schon früh Bestrebungen, einmal entwickelte und getestete Software für künftige Aufgaben wiederverwenden zu können. Hierbei besteht ein Zielkonflikt zwischen der aus der Spezifik einer Anwenderforderung folgenden Granularität der Software einerseits und der für die Wiederverwendbarkeit notwendigen Universalität andererseits.

In der Praxis der Softwareentwicklung sind häufig ähnliche Anforderungen zu realisieren, die auf Grund der nur auf bestimmte Sprachelemente beschränkten Flexibilität zu wiederum ähnlichen Programmen führen. Diese Ähnlichkeit zeigt sich beim Vergleich der Quelltexte dieser Programme in weitgehend identischen Passagen. Ein typisches Beispiel hierfür ist die in der kommerziellen Softwareentwicklung vielfach gewünschte Mandantenfähigkeit der Produkte. Dabei sollen gewisse Basisfunktionalitäten allen Anwendern zugänglich sein; einzelne Spezialfunktionen bedürfen jedoch einer kundenabhängigen Realisierung.

Kategorie	Flexibilität bzgl.
Unterprogramme	Werten von Variablen in Assemblersprachen
Funktionen	Werten von Variablen in imperativen Programmiersprachen
Templates	Datentypen in objektorientierten Programmiersprachen
Generative Programmierung	Struktur von Programmen unabhängig vom Paradigma

1. Einordnung

Das Konzept der generativen Programmierung überwindet die in Tabelle 1 genannten Grenzen, indem nicht einzelne Bestandteile der Sprache (Variablen, Datentypen usw.) sondern alle Elemente derselben problemabhängigen Änderungen unterworfen werden können. Somit lassen sich auch die an der Verarbeitung beteiligten Objekte (GUI, Filesystem, DBMS) sowie die darauf einwirkenden Algorithmen modifizieren. Während generative Programmierung bisher meist bei compilerbasierten¹ Sprachen zum Einsatz kam, sollen nachfolgend

Tabelle 1: Möglichkeiten der Flexibilisierung des Programmcodes

¹ C++ mit STL bei [Czarnecki/Eisenecker]

die Möglichkeiten dieser Technologie sprachübergreifend erprobt werden.

Zur formalen Beschreibung des Konzepts der generativen Programmierung eignet sich eine Kombination aus Mengenlehre und Prädikatenlogik:

```

program := statements ∪ variable
typeof(statement) = immutable
typeof(variable) = changeable
statements ∩ variable = ∅

```

Ein Programm setzt sich demnach zusammen aus der Beschreibung von zur Laufzeit veränderlichen und unveränderlichen Konstrukten und zerfällt in bzgl. dieser Eigenschaft disjunkte Teilmengen. Unabhängig von der Syntax der Programmiersprache beschreibt diese Semantikeigenschaft der Objekte unveränderbaren oder veränderbaren Speicherplatz. Zur Entwicklungszeit entstehen beide Objekttypen.

Betrachtet man zusätzlich den Zeitpunkt der Zugriffe auf die Objekte, so ist eine streng monotone Folge

$$t_0 < t_1 < \dots < t_{n-2} < t_{n-1} < t_n$$

zu erkennen, die Aktivitäten $\{create, modify, execute, delete\}$ ordnet: Activity = $f(t_i)$. Es sei

$create = f(t_{\geq 0})$	die Erzeugung eines Statements durch den Programmierer oder ein Programm
$modify = f(t_{>0})$	die Änderung der Semantik eines Statements nach seiner Erzeugung
$execute = f(t_{>0})$	die Interpretation der Semantik eines Statements zur Ausführzeit des Programms
$delete = f(t_n)$	die Löschung eines Statements als letztes Element der o.g. Folge

Tabelle 2: semantische Aktivitäten an Statements

In der Phase *execute* bezeichnet

$typeof(statement) = immutable$

z.B. ein Statement zur Ablaufsteuerung des Programms, das selbst jedoch keine Änderungen des von ihm belegten Speichers bewirkt. Dagegen führt

$typeof(variable) = changeable$

eine solche Änderung aus, z.B. durch Wertzuweisung an eine Variable.

Während in der konventionellen Programmierung gilt

$$\forall \text{statements, variable} \in \text{program: } f(t_0) = create \wedge f(t_{>1}) = execute \wedge f(t_n) = delete \wedge \nexists i \in t: f(t_i) = modify$$

ist die generative Programmierung gekennzeichnet durch

$$\forall \text{statements, variable} \in \text{program: } f(t_0) = create \wedge f(t_{>1}) = execute \wedge f(t_{\leq n}) = delete \wedge \exists i \in t: f(t_i) = modify$$

Der Existenzquantor definiert somit den generativen Ansatz: Solange die Menge der Statements in *program* ein diese Menge veränderndes Statement enthält, ist ein weiterer Generierungsschritt erforderlich. Dieser Prozess terminiert, sobald keine Variable mehr auftritt. Folglich ist die Generierung mehrstufig möglich.

2. Realisierung

Zur Realisierung von $f(i \in t) = modify$ kommen sprachspezifische Konstrukte zum Einsatz. Falls diese nicht oder nicht in ausreichendem Umfang zur Verfügung stehen, kann der Compiler oder Interpreter diese Aufgabe übernehmen. Beispiele hierfür gibt Tabelle 3.

Sprachklasse	Abstraktionsebene	Zeitpunkt	unterstützendes Sprachkonstrukt
Assembler	Maschinsprache	$f(t_{s_0})$	Befehlsmodifikation (z.B. EX-Befehl bei z/OS)
imperative Script-sprachen	Quelltext	$f(t_{s_0})$	Funktion <code>eval(...)</code> zur Ausdrucksauswertung oder <code>MSScriptControl</code> zur Ausführung von sprachfremdem Code
multiparadigmatische Sprachen	Quelltext	$f(t_{s_0})$	read-eval-print-loop z.B. in LISP
Beschreibungssprachen	Quelltext	$f(t_{s_0})$	Einbettung von Sprachen in HTML mit nachträglicher Erzeugung des Script-Tags
Office-Paket von Microsoft	Objekte	$f(t_{s_0})$	Erzeugen von Dokumenten (Word, Excel, PowerPoint, Access), die problemabhängigen Modifikationen unterliegen

Tabelle 3: Ausführbarkeit der generierten Instruktionsfolge

Vor der Illustration des Ansatzes durch ein Beispiel aus dem Office-Paket soll die formale Beschreibung des Konzepts stehen:

Ein Programm P ist eine Menge von Objekten s , die durch folgende Eigenschaften charakterisiert sind:

- n sei die laufende Positionsnummer innerhalb des Programms mit $n \in \mathbb{N}$.
- t sei Typ des Statements mit $t \in \{definition, function, class, control, assignment\}$.
- Jedes Element s der Menge P besitze einen eindeutigen Indexwert mit $i \in \mathbb{N}$.

Es sei $h(x, property)$ eine Hilfsfunktion, die den Index i des Elements als Argument akzeptiert sowie eine Zeichenkette $property$, die den Namen einer Objekteigenschaft angibt. Sie projiziert den so bestimmten Eigenschaftswert aus dem Objekt; ihr Rückgabewert ist *null*, falls die Eigenschaft nicht existiert.

Jedes Statement s besitzt eine eindeutige Positionsnummer.

$$\forall s_i: \exists! h(i, n) \neq h(j, n) \wedge s_i = s_j$$


Die Semantik der Programmabarbeitung legt fest, dass s_i zeitlich vor s_j ausgeführt wird, falls

$$i < j \wedge h(i, type) = assignment \wedge h(j, type) = assignment$$

Es habe jedes Statement eine Menge von charakteristischen Zeitpunkten, die ihrerseits eine monoton ansteigende Folge bilden. Damit verbundene Aktivitäten sind jene aus Tabelle 2 und können nun durch Verwendung der Index-Werte präzisiert werden.

Erstellung:	Liste der Zeitpunkte enthält genau ein Element, der Wert ist das kleinste Element aller Listen.
Abarbeitung:	Liste der Zeitpunkte enthält ein Element, wenn sich das Statement in einer Sequenz befindet. Sie hat mehr als ein Element, wenn es sich in einer Schleife befindet.
Modifikation:	Liste der Zeitpunkte ist leer bei der konventionellen Programmierung, andernfalls liegen sich selbst verändernde Programme vor.
Zerstörung:	Liste der Zeitpunkte hat genau ein Element und ist das größte Element aller Listen.

3. Beispiel

 Gedächtnistraining aus Ihrer Apotheke

Streichholzrätsel: Legen Sie ein einziges Streichholz so um, dass die Rechenaufgabe immer noch stimmt.

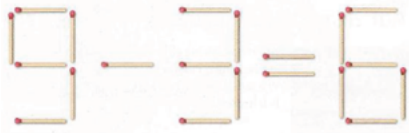


Abbildung 1: Diskursbereich²

Statt das Problem durch ein konventionelles Programm zu lösen, wird mit VBScript eine Generatorkomponente erstellt, die auf der Grundlage der ihr als Parameter übergebenen „Gleichung“ (vgl. Abbildung 1) eine Access-Datenbank angelegt und deren Tabellen problemspezifisch füllt.

² Quelle: „Kopf-fit“, Gedächtnistraining aus Ihrer Apotheke, S & D Verlag GmbH, Heft 6/2018, 38

Dies betrifft

- die Definition der Trägermenge einer Algebra (Menge der mit Streichhölzern darstellbaren „Zeichen“)
- die Definition der Operatormenge dieser Algebra (Teilmenge jener „Zeichen“, die als Operatorsymbole verwendbar sind)
- die Definition einer Menge von Elementen, die auf Grund ihrer Geometrie aus der Trägermenge ableitbar sind
- die Anzahl der dafür jeweils erforderlichen Bewegungen von Streichhölzern

Aus diesen in realen Tabellen gespeicherten Informationen gewinnt die generierte Komponente (hier also die Access-Datenbank) zur Laufzeit mit Hilfe virtueller Tabellen

- alle Ableitungsmöglichkeiten eines Zeichens (Abbildung 2)
- die Menge der daraus konstruier- und erfüllbaren Gleichungen (Abbildung 3)

```
SELECT derivation.symbol AS base, derivation.derivation AS modification,
purpose, alteration, 2-Abs(alteration) AS changed
FROM original INNER JOIN derivation ON original.symbol = derivation.symbol
UNION
SELECT derivation.derivation, derivation.symbol, purpose, alteration* -1, 2-
Abs(alteration)
FROM original INNER JOIN derivation ON original.symbol = derivation.symbol
UNION
SELECT symbol, symbol, purpose, 0, 0
FROM original;
```

Abbildung 2: Abfrage "capabilities" zur Erzeugung aller ableitbaren Symbole

Sie bildet die Grundlage der wie folgt erzeugten Gleichungen:

```
operand2.modification AS operand2, comparison.modification AS comparison,
result.modification AS result,
operand1.changed+operator.changed+operand2.changed+comparison.changed+result.
changed AS changes
FROM (SELECT * FROM capabilities WHERE purpose = True) AS operand1, (SELECT *
FROM capabilities WHERE purpose = False) AS operator, (SELECT * FROM
capabilities WHERE purpose = True) AS operand2, (SELECT * FROM capabilities
WHERE purpose = False) AS comparison, (SELECT * FROM capabilities WHERE
purpose = True) AS result
WHERE (((comparison.modification)='') AND ((operand1.base)='9') AND
((operator.base)='-') AND ((operand2.base)='3') AND ((comparison.base)='')
AND ((result.base)='6') AND
(((operand1).[alteration]+[operator].[alteration]+[operand2].[alteration]+[co
mparison].[alteration]+[result].[alteration])=0) AND
((Eval([operand1].[modification] & [operator].[modification] &
[operand2].[modification] & [comparison].[modification] &
[result].[modification]))<>False));
```

Abbildung 3: problemspezifische Generierung der Gleichungen

Die Funktion `Eval` kommt hier zum Einsatz, um die mit Variablenwerten aus der Datenbank belegte Gleichung auf ihre Erfüllbarkeit zu testen.

Die Präsentation des Ergebnisses übernimmt ein Report, der die zuvor erzeugten Abfragen ausführt und sich somit zu dessen Laufzeit die benötigten Informationen aus der Datenbank beschaffen kann.

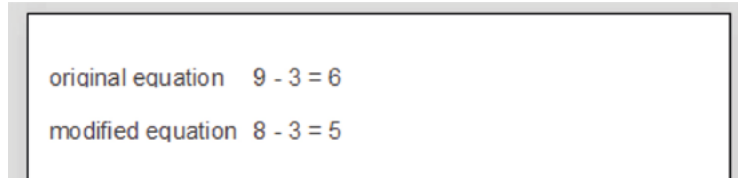


Abbildung 4: generierter Report

Den Report statet die Generierungskomponente mit Ereignisprozeduren aus, die wiederum zu einem späteren Zeitpunkt³ aktiv werden. In Abbildung 4 kommen sie zur Größenbestimmung und Positionierung der Felder zur Anwendung, die der Report aus dem ihm übergebenen SQL-SELECT-Statement ermittelt.

```
SELECT operand1 & " " & operator & " " & operand2 & " " & comparison & " " &
result AS equation,
Choose(Sgn(changes)+1,"original equation","modified equation") AS sense
FROM equations WHERE ((equations.[changes]) In (0,2))
ORDER BY equations.changes;
```

Abbildung 5: vom Generator an den Report übergebenes SQL-Statement

4. Fazit

Generative Programmierung kann die durch die jeweilige Programmiersprache bedingte Grenze der Flexibilisierung überwinden und erlaubt somit die anwenderspezifische Modifikation aller Konstrukte. Sie bedarf keiner zusätzlichen Software-Installation, sondern ist inzwischen mit den jeweiligen Bordmitteln realisierbar. Ein Einsatz ist vorteilhaft möglich, wenn

- Kundenanforderungen zur Aufteilung auf mehrere Softwareprodukte führen, die einander ähnlich sind und der ständigen Weiterentwicklung bedürfen
- das DRY-Prinzip ("Don't repeat yourself") über die Grenzen eines Systems hinweg durchgesetzt werden muss (z.B. bei der Erstellung von Datenbankobjekten und deren späterer Nutzung in 3GL oder 4GL)
- erst zur Laufzeit eines Programmes Bedingungen entstehen, die in nachfolgenden Schritten berücksichtigt werden müssen (dynamische Constraints)
- Code anwendungsspezifisch bzgl. seiner Ausführzeit optimiert werden muss (z.B. Ersetzen von Variablen durch Konstanten)
- Office-Dokumente einer kundenspezifischen Gestaltung bedürfen (Ereignisbehandlung)

Insbesondere das Office-Paket bietet sich zur Realisierung der in Tabelle 3 vorgestellten Prinzipien an:

- Werden mehrere Dokumente mit zueinander ähnlicher Funktionalität benötigt, so kann ein aus VBA oder VBScript bestehender Generator diese erzeugen.
- Das kundenspezifische Generieren von Ereignisbehandlungsprozeduren, SQL-Abfragen sowie Formularen und Berichten ermöglicht einen mehrstufigen Prozess, da diese Objekte selbst wieder die Funktion eines Generators übernehmen können. Analoges gilt für HTML im Zusammenhang mit den dort einzubettenden Sprachen.
- Die konsequente Anwendung dieses Verfahrens ermöglicht die sprach- bzw. systemübergreifende Durchsetzung von Integritätsregeln – insbesondere bzgl. der Benennung von Objekten, die in unterschiedlichen Umgebungen verwaltet werden (z.B. Namen von Datenbanken, Tabellen, Abfragen, Oberflächenelemente und Programmcode, der auf diese zugreift).

³ beim Öffnen des Reports sowie während der Formatierung seines Detailbereiches

Dem Vorteil einer nahezu unbegrenzten Flexibilisierung stehen Nachteile gegenüber:

- Die Entwicklung der Generatorkomponente ist mit – im Vergleich zu konventionellen Ansätzen – zusätzlichem Aufwand verbunden. Die Wirtschaftlichkeit ist folglich nur dann gegeben, wenn die Erzeugung von Derivaten einer Lösung entsprechend häufig gefordert wird.
- Die Prüfung der formalen Korrektheit der Programme wird erschwert; jedoch ist diese besser beweisbar, wenn die formale Korrektheit des Generators bewiesen werden kann.

Der letztgenannte Nachteil relativiert sich, wenn die Weiterentwicklung der „ähnlichen“ Programme der konventionellen Vorgehensweise in die Betrachtung einbezogen wird: die dort bestehende Redundanz der in den Algorithmen formulierten Regeln führt leicht zur Inkonsistenz, wenn nicht in allen Varianten semantisch äquivalente Änderungen vollzogen werden. Dieses Problem löst der generative Ansatz, indem eine Änderung nur noch einmal in der Generatorkomponente erforderlich ist und sich diese dann automatisiert in die generierten Komponenten überträgt.

Literatur

Czarnecki, Krzysztof; Ulrich W. Eisenecker (2000): Generative Programming: Methods, Tools, and Applications. Addison Wesley.

Klar, Michael (2006): Einfach generieren: Generative Programmierung verständlich und praxisnah. München: Hanser Fachbuchverlag.

Magazin „Kopf-fit“, S & D Verlag, Geldern, Ausgabe Heft 6/2018

Sporer, Mathias (2019): Lehrmaterialien für Konzepte der generativen Programmierung. In "Wissen im Markt", wissenschaftliche Zeitschrift der Berufsakademie Sachsen, Ausgabe 2019, 20 ff.

