4

Prinzipien der Codegenerierung

Der Worte sind genug gewechselt,
lasst mich auch endlich Taten sehen.
Faust, Johann Wolfgang von Goethe

Codegenerierung ist ein wesentlicher Erfolgsfaktor für den Einsatz von Modellen im Softwareentwicklungsprozess. Aus vielen Modellen kann Code für das Produktionssystem oder für Testtreiber effizient generiert und damit die Konsistenz zwischen Modell und Implementierung verbessert sowie Ressourcen eingespart werden. Dieses Kapitel beschreibt grundlegende Konzepte, Techniken und Probleme der Codegenerierung und skizziert eine Darstellungsform für Regeln zur Codegenerierung in Form von Transformationsregeln.

4.1 Konzepte der Codegenerierung 76
4.2 Techniken der Codegenerierung 85
4.3 Semantik der Codegenerierung 92
4.4 Flexible Parametrisierung eines Codegenerators 94

B.Rumpe, *Agile Modellierung mit UML*, 2. Aufl., Xpert.press
DOI 10.1007/978-3-642-22430-0_4, © Springer-Verlag Berlin Heidelberg 2012

Die Möglichkeit, aus einem Modell ablauffähigen Code zu erzeugen, bietet interessante Perspektiven bei der Softwareentwicklung und ist teilweise sogar eine wesentliche Voraussetzung für

- die Steigerung der Effizienz der Entwickler [SVEH07],
- die Trennung von Anwendungsmodellierung und technischem Code, die die Wartbarkeit und die Weiterentwicklung der Funktionalität sowie die Portierung auf neue Hardware und Betriebssystemversionen besser unterstützt [SD00],
- Rapid Prototyping mit Hilfe von Modellen, die eine kompaktere Beschreibung des Systems erlauben, als es durch eine reine Programmiersprache wie Java möglich wäre,
- schnelles Feedback durch Demonstrationen und Testläufe und
- einen wesentlichen Aspekt der Qualitätssicherung: der Generierung von automatisierten Tests.

Eine der Stärken der Generierung ist die schnelle Erstellung sich häufig wiederholender ähnlicher Codefragmente (oder „Aspekte“, [LOO01, KLM+97]). Gerade bei der Anbindung technischer Aspekte, wie etwa GUI, Peristenz oder Kommunikation verteilter Systemteile sind häufig struktuell gleich und können sehr gut aus abstrakten Modellen abgeleitet werden. Das reduziert drastisch die Größe der manuell zu erstellenden Artefakte. Dies wiederum führt zu weniger Programmierfehlern, größerer Konformität des generierten Codes zu Codierungsstandards[1] sowie zur schnelleren Erstellung und flexibleren Anpassung des Systems auf Basis von Modellen.

Probleme heutiger Werkzeuge

Die Erzeugung von ablauffähigem Code aus einem Modell ist daher derzeit zurecht eine der wesentlichen Anstrengungen der Hersteller von Modellierungswerkzeugen. Dies gilt nicht nur für UML-basierte Werkzeuge, sondern auch für Werkzeuge für ähnliche Sprachen, wie Autofocus [HSSS96, Sch04], der in der Telekommunikation verwendeten SDL [IT07b, IT07a] oder den in Statemate und Rhapsody umgesetzten Statecharts [HN96]. Aufgrund dieser vielfältigen Anstrengungen ist davon auszugehen, dass sich die Situation bei der Codegenerierung in den nächsten Jahren weiter verbessern wird.

Viele der heute existierenden Werkzeuge bieten bereits die Generierung von Code oder Coderahmen aus Teilen der UML an[2]. Jedoch gibt es hier eine Reihe grundsätzlicher Probleme, die auch konzeptioneller Verbesserungen bedürfen.

1. Die Erzeugung von Coderahmen aus Klassendiagrammen ist mittlerweile Stand der Technik. Dabei werden Hüllen für die Klassen erzeugt, die

[1] Das ist zum Beispiel bei einer Zertifizierung des Systems wichtig.
[2] Ein Überblick über Generatoren ist in umltools.net zu finden.

zumindest Attributdefinitionen und Zugriffsfunktionen beinhalten. Die Rümpfe generierter Methoden sind manuell einzutragen. Da detaillierte Modelle im Projekt meist einer hohen Änderungsrate unterworfen sind, müssen die manuell eingesetzten Coderümpfe nach jeder Generierung neu nachgetragen werden oder gehen verloren. Als Ausweg wird deshalb auch „Roundtrip-Engineering" [SK04] verwendet.

2. Roundtrip-Engineering erlaubt die wechselseitige Transformation von Code in Klassendiagramme und umgekehrt. Wesentlich ist dabei, dass beide Sichten manuell änderbar sind, ohne dass die Änderungen in der jeweils anderen Sicht verloren gehen. Insbesondere bleiben Methodenrümpfe in der Code-Sicht erhalten auch wenn sie im Klassendiagramm nicht sichtbar sind. Wenn aber eine möglichst kompakte Darstellung des Systems gewünscht ist, dann ist das eine Sackgasse. Sinnvoller ist es dann nur eine Darstellung anzubieten, die graphische Klassendiagramme und Coderümpfe integriert. Die wesentlichen Hindernisse dafür sind das derzeit noch zu geringe Zutrauen des Entwicklers in den generierten Code, so dass ein manueller Eingriff in den generierten Code noch gewünscht wird, und die nicht zufriedenstellend geklärte Frage, wie und wo Coderümpfe abgelegt werden, so dass sie vom Entwickler effizient bearbeitet werden können.
   Mit den ersten Compilern war die Situation jedoch ähnlich. Es wurde Assembler-Quellcode erzeugt, der manuell änderbar sein sollte. Es kann davon ausgegangen werden, dass mit zunehmender Reife der Generierungstechnologie die Ebene des lesbaren Quellcodes unwichtiger wird und Bytecode direkt erzeugt werden kann. Dann wird es auch nicht notwendig sein, dass der generierte Code *Coding Guidelines* erfüllt und gut lesbar ist.
3. Werden Coderümpfe wie beim Roundtrip-Engineering direkt in den generierten Code eingesetzt, so ist in diesen Coderümpfen kaum mehr eine Abstraktion von der konkreten Realisierung von Attributen, Assoziationen, etc. möglich. Stattdessen muss der Entwickler die Form der Umsetzung und die daraus resultierenden Zugriffsfunktionen kennen. Werden Coderümpfe aber ebenfalls generiert, so können zum Beispiel Attributzugriffe durch entsprechende `get`- und `set`-Methoden ersetzt werden. Auch ist dann die Instrumentierung des Codes für Tests besser möglich.
4. Leider stimmt die dokumentierte oder den Analyse- und Refactoring-Techniken zugrunde liegende Semantik (im Sinne von Bedeutung, [HR04]) mit dem bei der Codegenerierung entstandenen Verhalten gelegentlich nicht überein. Dies ist ein generelles Problem , das einer sorgfältigen Festlegung von Codegenerierung, Analysen, Refactoring-Techniken und der dokumentierten Semantik bedarf. Denn sonst ist es möglich, dass ein bezüglich der festgelegten Semantik nachweislich korrektes Refactoring doch zu einem veränderten Systemverhalten führt.
5. Einfache Werkzeuge generieren oft eine starre Form von Code, ohne auf die spezifischen Bedürfnisse des Projekts einzugehen. Eine

Parametrisierung der Codegenerierung ist wünschenswert und an vielen Stellen auch notwendig, um zum Beispiel plattformspezifische Anpassungen in den Code einzubauen, die Nutzung von Frameworks, Speicher- und Kommunikationstechniken zu erlauben oder die Optimierung der Umsetzung von Modellelementen zu ermöglichen.
Die möglichen Formen der Codegenerierung sind vielfältig und können nicht direkt antizipiert werden. Deshalb ist einerseits eine flexible Template- oder Skriptsprache[3] notwendig, andererseits dennoch zu sichern, dass die „essentielle Semantik" der Diagramme durch die technologiespezifische Codegenerierung nicht verloren geht.

In diesem Kapitel werden grundlegenden Konzepte zur Codegenerierung diskutiert und ausschnittsweise anhand der UML/P-Notation erläutert. Für ein weitergehendes Studium von Gererierungstechniken wird auch [CE00] empfohlen. Der Abschnitt 4.1 erörtert Konzepte von Codegeneratoren, die auch die im Kapitel 7 diskutierte Verwendung der UML zur Modellierung von Testfällen behandelt. In Abschnitt 4.2 werden Anforderungen wie Flexibilität, Plattformunabhängigkeit und Steuerbarkeit eines Codegenerators diskutiert. Abschnitt 4.3 behandelt die Beziehung zwischen Codegenerator und Semantikdefinition der Sprache. Abschnitt 4.4 beschreibt, wie ein flexibler Codegenerator aussehen kann. Ein solcher flexibel anpassbarer Generator für die UML/P steht mit [Sch12] zur Verfügung.

4.1 Konzepte der Codegenerierung

In Vorwegnahme der nachfolgend diskutierten konzeptionellen Grundlagen werden in Abbildung 4.1 die wesentlichen Begriffe einführend definiert.

Der Einsatz eines Codegenerators hat einige Vorteile gegenüber konventioneller Programmierung. Die *Verständlichkeit* der benutzten Modellierungsbeziehungsweise Programmiersprache wird erhöht, indem die Sprache kompakter und/oder durch graphische Elemente übersichtlicher wird. Die *Effizienz der Softwareentwicklung* wird erhöht. Allein dadurch, dass weniger Code manuell zu schreiben, prüfen und testen ist, können Entwickler ihre Effizienz steigern. Zusätzliche Aspekte, wie die bessere *Wiederverwendbarkeit* von abstrakten Modellen aus einer Modellbibliothek, steigern die Entwicklereffizienz weiter. Dies führt zu einer Reduktion des Gesamtaufwands für die Softwareentwicklung. Dadurch wird weniger Projektorganisation notwendig, wodurch weitere Effizienzsteigerungen möglich werden.

Die Wiederverwendbarkeit ist dabei auf mehreren Ebenen möglich. Ein Modell kann in angepasster Form in einem ähnlichen Projekt oder einer Produktlinie [BKPS04] wiederverwendet werden. Idealerweise kann durch wiederholte Verbesserung ein *Modell-Framework* entstehen, das für gleichartige

[3] „Assistenten" oder „Wizards" bieten zusätzlich Werkzeugunterstützung bei der Skripterstellung.

Konstruktives Modell ist eine Spezifikation des Systems, die mit Hilfe eines automatischen Generators zur Codegenerierung eingesetzt wird. Die Veränderung eines konstruktiven Modells hat die direkte Veränderung des Produkts zur Folge. Die Modellierungssprache wird auch als High-Level-Programmiersprache angesehen, da sie grundsätzlich ausführbar ist.

Deskriptives Modell ist eine Spezifikation, die zur Beschreibung des Systems verwendet wird, ohne konstruktiv bei der Implementierung eingesetzt zu werden. Das sind typischerweise abstrakte und unvollständige Beschreibungen, also insbesondere Modelle, die als Vorlage für eine manuelle Implementierung dienen oder erst nach Systemerstellung als Dokumentation verfasst werden.

Testmodell ist eine Spezifikation, die für die manuelle oder automatische Ableitung von Tests geeignet ist. Das Testmodell wird in ausführbaren Code übersetzt, der zum Aufbau von Testdatensätzen, als Testtreiber oder als Test-Sollergebnis eingesetzt wird.

Konstruktives Testmodell ist ein Testmodell, das durch den Codegenerator in ausführbare Tests übersetzt wird. Demgegenüber werden deskriptive Testmodelle manuell in Tests übersetzt.

Codegenerierung ist der Vorgang zur Erzeugung von Code aus konstruktiven Modellen.

Codegenerator ist ein Programm, das ein konstruktives Modell einer höheren Programmiersprache in eine Implementierung transformiert (nach [CE00]). Der generierte Code kann zum Produktionssystem oder zum Testcode gehören.

Skript beinhaltet die konstruktive Steuerung der Codegenerierung. Skripte parametrisieren den Codegenerator und erlauben damit plattform- und aufgabenspezifische Codegenerierung.

Template ist eine spezielle Form eines Skripts. Ein Template beschreibt Codemuster, in die bei der Generierung konkrete Elemente des Modells eingesetzt werden. Dabei wird typischerweise ein Makro-Ersetzungsmechanismus verwendet.

Abbildung 4.1. Begriffsdefinitionen zur Codegenerierung

Projekte direkt verwendbar ist und sogar einen speziell dafür geeigneten Codegenerator besitzt. Das im Codegenerator eingebettete technische Wissen beispielsweise zur Erzeugung von Schnittstellen oder sicherer und effizienter Übertragungsmechanismen kann unabhängig davon wiederverwendet werden. Eine weitere Möglichkeit zur Wiederverwendung von Modellen ergibt sich innerhalb eines Projekts. Ein Objektdiagramm kann zum Beispiel sowohl als Prädikat als auch konstruktiv zur Erzeugung einer Objektstruktur eingesetzt werden. Beide Formen können außerdem im Produktionssystem oder bei der Testfalldefinition eingesetzt werden. Der dafür generierte Code ist, wie in Abschnitt 5.2 noch diskutiert, sehr unterschiedlich und daher manuell viel aufwändiger zu erstellen.

Gelegentlich sind Codegeneratoren heute auch bereits in der Lage, *effizienteren Code* zu erstellen, als dies in vertretbarem Aufwand durch manuelle Optimierungen möglich wäre. Dies gilt natürlich vor allem für ausgereifte Compiler normaler Programmiersprachen, die eine Reihe von

Optimierungstechniken einsetzen. Für ausführbare Modellierungssprachen wie die UML/P ist davon auszugehen, dass die Steigerung der Effizienz der Entwickler derzeit durch eine weniger effiziente Implementierung erkauft werden muss. Entsprechend ist der Einsatz von Generatoren für Modellierungssprachen vor allem bei Individualsoftware und erst in zweiter Linie bei eingebetteter, massenhaft in möglichst kostengünstigen Geräten vertriebener Systemsoftware sinnvoll.

Die flexible Generierung von Code aus fachlichen Modellen erlaubt letztendlich auch die Behandlung von technischen und teilweise fachlichen Variabilitäten im Sinne von [HP02]. Dabei werden offene technische Aspekte des funktionalen Modells durch einen jeweils technologiespezifisch angepassten Generator geeignet ausgefüllt.

4.1.1 Konstruktive Interpretation von Modellen

Wie bereits in Band 1 beschrieben, ist ein *Modell* seinem Wesen nach eine in Maßstab, Detailliertheit oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems [Sta73]. Modelle werden immer dort eingesetzt, wo das tatsächliche System so komplex ist, dass es sich zunächst lohnt, bestimmte Eigenschaften des Systems am Modell zu analysieren oder dem Kunden zu erklären. Dazu gehören Architekturmodelle von Gebäuden ebenso wie technische Modelle komplexer Maschinen oder Modelle sozialer und wirtschaftlicher Zusammenhänge. Bei manchen Modellen, wie zum Beispiel Bauplänen oder Schaltzeichnungen, steht der Wunsch nach einer Beschreibung des Aufbaus (Architektur) im Vordergrund, bei anderen die Simulation von Funktionalität und anderer verhaltensorientierter Eigenschaften.[4]

Generell gilt aber, dass diese Modelle und Bauzeichnungen als Hilfsmittel für die spätere Erstellung des Artefakts dienen. Wird das Modell erstellt, um *danach* das eigentliche Artefakt zu bilden, so hat das Modell eine *vorschreibende* (*präskriptive*) Wirkung. Im Gegensatz dazu wird ein Modell *beschreibend* (*deskriptiv*) eingesetzt, wenn das Original vor dem Modell existiert. Beispiele hierzu sind etwa eine Modelleisenbahn oder Fotografien [Lud02].

Aufgrund der Immaterialität von Software entfalten Modelle in der Softwareentwicklung zusätzlich zur deskriptiven Wirkung auch eine *konstruktive* Wirkung. Wenn für die Generierung lauffähiger Software aus einem immateriellen, im Computer gespeicherten Modell nur ein Knopfdruck notwendig ist, dann wirkt das Modell als konstruktive Vorgabe. Der Quellcode einer Programmiersprache kann aufgrund der automatisierten Übersetzung als zu dem erzeugten Objektcode äquivalent angesehen werden. Streng genommen sind Quellcode und Objectcode ebenfalls Modelle des Systems. Für praktische Belange werden sie jedoch – und mit Recht vereinfachend – mit

[4] Für eine tiefergehende Diskussion des Modellbegriffs eignen sich zum Beispiel [Sta73] oder [Lud02].

dem System selbst identifiziert. Dieselbe Annahme kann auch für ausführbare UML/P-Modelle getroffen werden.

Die konstruktive Verwendung der Modelle hat einige Auswirkungen, die bei einem nicht-konstruktiven Einsatz nicht auftreten. Zum Beispiel verändert die Hinzunahme oder das Weglassen von Elementen des Modells sofort das modellierte System. Ein Beispiel ist die Verwendung mehrerer Statecharts zur Modellierung des Verhaltens einer Klasse auf verschiedenen Abstraktionsstufen. Ein Generator, der damit umgehen kann, simuliert diese parallel und realisiert damit ein mehrdimensionales Zustandskonzept[5] für eine Klasse.[6] Wird nun ein weiteres Statechart als Modell hinzugenommen, das ausschließlich bereits vorhandene Information in abstrakterer Form darstellt, so wird das Zustandskonzept weiter aufgebläht. Das ändert zwar nicht das funktionale Gesamtverhalten, wohl aber die interne Struktur und das Zeitverhalten des Systems.

Modelle werden also in der Softwareentwicklung in verschiedenen Rollen eingesetzt. Dazu gehören die automatisierte Generierung von Produktionscode, aber auch von Tests. Die manuelle Umsetzung eines Modells ist ebenso möglich, wie die Erstellung von Modellen, nachdem das Artefakt bereits existiert. Abbildung 4.2 charakterisiert die drei Dimensionen zur Unterscheidung des Einsatzes von Modellen.

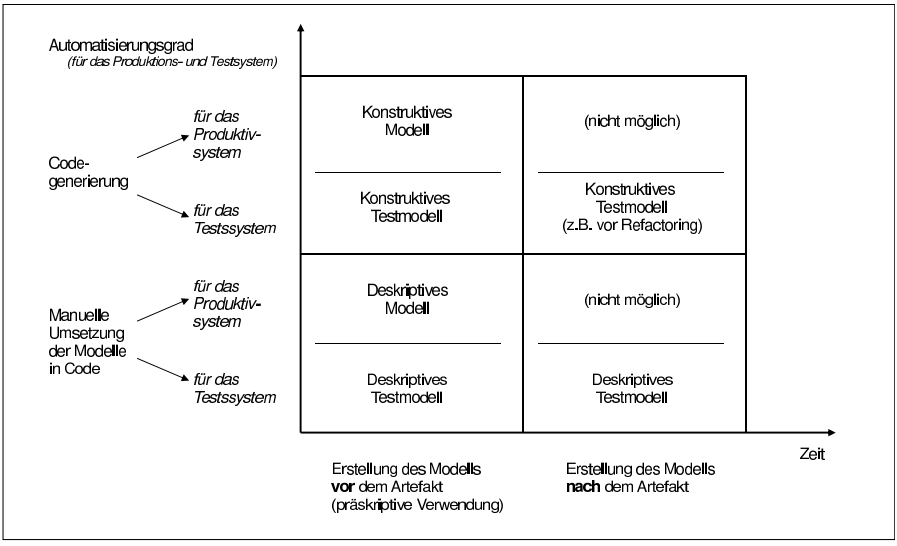


Abbildung 4.2. Varianten des Einsatzes von Modellen

[5] In der Regel sind das Kreuzprodukte.

[6] Dabei wird davon ausgegangen, dass Werkzeuge nicht automatisch in der Lage sind zu erkennen, dass ein Statechart eine Abstraktion eines anderen ist.

Streng genommen ist die Erstellung eines konstruktiv verwendeten Modells nach dem Artefakt möglich und zum Beispiel im Reverse Engineering sinnvoll. Jedoch wird dieses Modell nicht zur Erstellung des bereits vorhandenen Originals, sondern für die nächste Version verwendet.

Der Unterschied zwischen der konstruktiven und der deskriptiven Interpretation von Modellen ist verwandt zu einem ähnlichen Phänomen, das bei algebraischen Spezifikationssprachen detailliert diskutiert wurde. Einem deskriptiv eingesetzten Modell sollte eine *lose Semantik* [BFG+93] zugeordnet sein. Das heißt, dass verschiedene Implementierungen beschrieben werden können, die das Modell erfüllen. Viele dieser Implementierungen enthalten beispielsweise weitere Zustandskomponenten, Funktionalität oder Schnittstellen, die im vorliegenden, unvollständigen Modell nicht explizit erwähnt sind. Ein Klassendiagramm beschreibt dann einen Ausschnitt eines Systems, da noch weitere, ungenannte Klassen besitzen kann. Ein deskriptives Modell kann daher unvollständig sein. In Band 1 [Rum11] wurde beispielsweise eine lose Semantik für Sequenzdiagramme beschrieben.

Demgegenüber stellt ein konstruktiv eingesetztes Modell eine vollständige Beschreibung des Softwaresystems dar, da allein aus dem Modell das gesamte lauffähige System generiert wird. Dies entspricht bei algebraischen Spezifikationen einer *initialen Semantik*, die einem Modell genau eine Implementierung zuordnet.[7,8]

4.1.2 Tests versus Implementierung

Die UML/P erlaubt die Erstellung von Modellen, die sich sowohl zur Testgenerierung als auch zur Generierung für das Produktionssystem eignen. Dazu zählen Objektdiagramme, die, wie in Abschnitt 4.4, Band 1 besprochen, als Vorbedingungen konstruktiv eingesetzt werden, um die Ausgangssituation eines Tests herzustellen, und als Nachbedingungen eingesetzt werden, um zu beschreiben, welche Situation nach Anwendung der Funktion für einen Testerfolg erfüllt sein muss.

Aus bestimmten Teilen eines Modells lässt sich auch kein konstruktiver Code, sondern nur Testcode generieren. Beispielsweise sind OCL-Bedingungen im Allgemeinen ausführbar. Wie in Abschnitt 3.3.10, Band 1 beschrieben, gilt dies meist auch bei Benutzung von Quantoren, da mit Ausnahme der Quantoren über Grunddatentypen wie `int` und mengen- beziehungsweise listenwertigen Typen höheren Grades alle Quantoren endlich und damit auswertbar sind.

Dennoch ist es ein wesentlicher Unterschied, ob eine in OCL formulierte Nachbedingung nur getestet oder sogar konstruktiv erzwungen werden

[7] In der Theorie algebraischer Spezifikationen werden Modelle als „Spezifikationen“ und Elemente des semantischen Domains beziehungsweise Implementierungen als „Modelle“ bezeichnet.

[8] Um eine initiale Semantik zu besitzen, müssen bestimmte Einschränkungen an die Form der Spezifikationen gestellt werden. Siehe dazu auch [EM85].

kann. Nahezu alle praktisch interessanten OCL-Bedingungen fallen in die erste Kategorie. Die Kategorie der konstruktiven OCL-Bedingungen ist allerdings deutlich kleiner. Das demonstrieren die folgenden zwei Beispiele.

Sortieren

Die Methode `sort` soll ein Array von Zahlen (`int`) sortieren. Eine öfter zu findende Beschreibung in Form einer Vor-/Nachbedingung ist die Folgende:

```
context int[] sort(int a[])                                    OCL
pre:  true
post: forall int i in {1..result.length-1}:
        result[i-1] <= result[i]
```

Diese Spezifikation kann sehr einfach und in linearer Zeit getestet werden. Als konstruktive Beschreibung ist sie allerdings nicht geeignet, weil ein Generator daraus keinen Sortieralgorithmus erzeugen kann. Darüber hinaus ist sie in wesentlichen Eigenschaften unvollständig, da sie nicht sichert, dass die Ausgangselemente der Reihung `a[]` in der Ergebnisreihung `result[]` wieder vorkommen müssen. Tatsächlich wäre daher eine Implementierung der Form `result=new int[0]` ebenfalls korrekt.

Die konstruktive Beschreibung für einen Sortieralgorithmus ist zwar im Prinzip möglich, aber genauso komplex wie eine direkte Implementierung. Gerade bei komplexen Algorithmen zeigt sich der wesentliche Vorteil deskriptiver Beschreibungen, da sie keine Implementierungsform vorwegnehmen. Sie sind daher insbesondere gegenüber effizienten Implementierungen sehr viel leichter verständlich.

Gleichungen als Zuweisungen

Zuweisungsmethoden haben im Allgemeinen die einzige Aufgabe, das möglicherweise gekapselte Attribut zu setzen:

```
context void setAttr(Type val)                                 OCL
pre:  true
post: attr==val
```

Diese Spezifikation ist sowohl für Tests der Methode `setAttr` als auch für eine konstruktive Umsetzung in eine Implementierung geeignet. Wird nämlich der Gleichheitsoperator == durch den Java-Zuweisungsoperator = ersetzt, so kann die Nachbedingung als Implementierung verwendet werden. Diese Implementierung ist allerdings nur dann wirklich korrekt, wenn keine weiteren Invarianten existieren, die eine zusätzliche Veränderung anderer Attribute erforderlich machen.

Leider ist die konstruktive Umsetzung von Nachbedingungen nur unter bestimmten, sehr eng umrissenen Rahmenbedingungen möglich. Typischerweise darf eine Nachbedingung nur aus einer Konjunktion von Zuweisungen an lokale Variablen bestehen und spätere Zuweisungen dürfen die

früheren nicht wieder invalidieren. Beispielsweise ist `val==attr` zur obigen Nachbedingung äquivalent, kann aber in dieser Form nicht in Code umgesetzt werden.[9] Auch die nachfolgende Bedingung ist für eine Codegenerierung ungeeignet, da sie zyklische Abhängigkeiten enthält:

```
context void method(Type val)
pre:  true
post: a==b+1 && b==2*a-val
```

Für ihre konstruktive Umsetzung ist zunächst das lineare Gleichungssystem zu lösen und es kann konstruktiv formuliert werden:

```
context void method(Type val)
pre:  true
post: a==val-1 && b==val-2
```

Natürlich gibt es eine Reihe trickreicher Verfahren zur konstruktiven Interpretation von Bedingungen, die für verschiedene Hochsprachen entwickelt wurden. Von denen seien insbesondere die Horn-Klausel-Logik von Prolog [Llo87], die Auswertung von in Gleichungslogik formulierter algebraischer Spezifikationen [EM85] und deren Erweiterung um Konditionale erwähnt. Beispielsweise kann auch die folgende Spezifikation konstruktiv umgesetzt werden:

```
context int abs(int val)
pre:  true
post: if (val>=0) then result==val else result==-val
```

Arten der Generierung

Da die Notationen der UML/P zur Modellierung von exemplarischen und vollständigen Strukturen und Verhalten eingesetzt werden, eignen sie sich in unterschiedlicher Weise zur Generierung von Code. Abbildung 4.3 zeigt, welche Diagrammart hauptsächlich (dicker Pfeil) und nebenbei (dünner Pfeil) für welche Form der Code- beziehungsweise Testgenerierung eingesetzt wird. Es ist aber festzuhalten, dass sich nicht alle Konzepte der UML/P-Dokumente zur Codegenerierung eignen. UML/P erlaubt grundsätzlich die Abstraktion von Details, zum Beispiel durch Auslassen von Typinformation bei Attributen oder durch Unterspezifikation bei Methoden und Transitionen, so dass die Fähigkeit zur Code- und Testgenerierung aus einem UML/P-Artefakt unter anderem von dessen Vollständigkeit abhängt. Intelligente Generierungsalgorithmen können natürlich auch unvollständige Artefakte zur Generierung nutzen, indem sie die offenen Aspekte durch Defaults ausfüllen

[9] Genau genommen ist `val=attr` in Java erlaubt, hat jedoch einen anderen Effekt als gewünscht.

oder intelligent raten. So kann zum Beispiel bei einem unvollständigen Statechart ein standardmäßiges Fehlerverhalten hinzugefügt werden und bei Attributen ohne Typinformation versucht werden, durch Typinferenz an den Stellen der Attributnutzung den benötigten Typ auszurechnen.

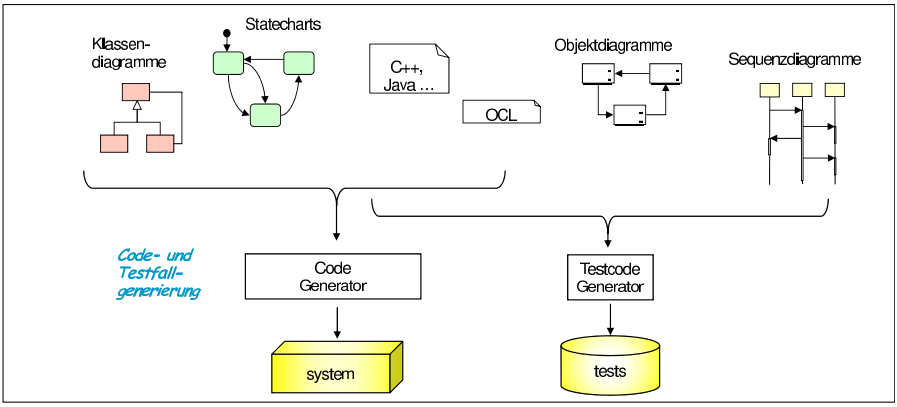


Abbildung 4.3. Generierung von Code und Tests aus UML/P

Der generierte Produktionscode kann dabei je nach Verwendungszweck mit zusätzlichem Testcode instrumentiert sein. So können für Testzwecke Inspektionsmethoden, interaktive Haltepunkte, Funktionen zum Zugriff auf private Attribute oder die Prüfung von Invarianten in den Produktionscode integriert sein, die bei der Erzeugung des Produktionscode für den Einsatz als fertiges Produkt weggelassen werden. Diese Form der Instrumentierung birgt Probleme, wenn der optionale Code Seiteneffekte beinhaltet, die das Verhalten des instrumentierten Produktionscodes verändern. Es ist deshalb wichtig, dass eine solche Instrumentierung nicht manuell, sondern von Codegeneratoren durchgeführt wird, so dass verhaltensverändernde Seiteneffekte ausgeschlossen werden können. Die durch die Instrumentierung entstandene Veränderung des zeitlichen Verhaltens muss in nebenläufigen Systemen unter gesonderten Gesichtspunkten betrachtet werden.

4.1.3 Tests und Implementierung aus dem gleichen Modell

Wie im vorherigen Abschnitt diskutiert, lassen sich aus manchen Modellen einerseits Tests, andererseits aber auch konstruktiver Code generieren. Die Generierung beider Codearten aus demselben Modell kann jedoch kein zusätzliches Vertrauen in die Richtigkeit des erstellten Systems erzeugen. Werden aus einem falschen Modell sowohl fehlerhafter Implementierungscode erzeugt als auch die Tests für diese Implementierung abgeleitet, so sind die Tests in gleicher Weise falsch. Dies zeigt das folgende einfache Beispiel, das den Absolutwert einer Zahl berechnen soll:

```ocl
context int abs(int val)
pre:  true
post: result==-val
```

Die Codegenerierung kann damit folgenden Java-Code erstellen:

```java
int abs(int val) {
  return -val;
}
```

Eine typische Sammlung von Tests benötigt mehrere Eingabewerte, auf denen getestet wird. Als gute Standardwerte haben sich für den Datentyp `int` Sammlungen von Zahlen der Form $-n, -2, -1, 0, 1, 2, n$ für einige große n herausgestellt.[10] Normalerweise werden diese vom Entwickler vorgegeben. Die erwarteten Ergebnisse müssen nicht separiert ausgerechnet werden, da mit der Nachbedingung eine Möglichkeit zur Prüfung der Korrektheit des Ergebnisses existiert. Folgender Testcode würde erzeugt werden können:

```java
int val[] = new int[] {-1234567,-2,-1,0,1,2,3675675};
for(int i = 0; i<val.length; i++) {
  int result = abs(val);
  ocl result==-val;
}
```

Da der Testcode genauso falsch ist wie die Implementierung, würde der Fehler damit nicht erkannt werden. In so einer Situation wird eigentlich nicht der implementierte Code getestet, sondern es wird getestet, ob der Codegenerator korrekt funktioniert. Denn wenn in dieser Situation ein Fehler gemeldet werden würde, dann würde der nur auf eine Inkonsistenz zwischen dem generierten Code und dem ebenfalls generierten Testtreiber hinweisen. Ein solches Vorgehen ist genau dann interessant, wenn die Parametrisierung des Generators getestet werden soll.

Als Konsequenz dieser Beobachtung ergibt sich, dass das konstruktive, zur Codegenerierung verwendete Modell und das Testmodell getrennt modelliert werden müssen. Dabei dürfen Fragmente des Test- und des konstruktiven Modells in denselben Diagrammen dargestellt sein. Es ist jedoch klar zu trennen, welche Konzepte wofür verwendet werden. Beispielsweise werden Statecharts im Wesentlichen konstruktiv eingesetzt. Die in den Statecharts verwendbaren Zustandsinvarianten werden jedoch bis auf Ausnahmen nur zur Prüfung in Tests eingesetzt.

[10] Der Auswahl der Zahlen liegt eine einfache Klassifikation für den Wertebereich `int` und die Verwendung einzelner Vertreter und Grenzwerte aus den gebildeten Klassen zugrunde.

4.2 Techniken der Codegenerierung

4.2.1 Plattformabhängige Codegenerierung

Obwohl mit der Festlegung der UML/P auf die Programmiersprache Java bereits eine wesentliche Entwurfsentscheidung getroffen wurde, ist die Form des generierten Codes nicht eindeutig festgelegt. Es gibt mehrere Dimensionen von Variationen, die bei der Codegenerierung zu beachten sind. Dazu zählt zum einen die hier besprochene Plattformabhängigkeit, die besonders bei eingebetteten Systemen eine wesentliche Rolle spielt.

Je nach Zielplattform stehen unterschiedliche Mechanismen zur Verfügung, um zum Beispiel Kommunikation im verteilten System mit den gesteuerten Anlagen, Nachbarsystemen, der Cloud oder Nutzern sowie Speicherung und Fehlerbehandlung durchzuführen oder die Einbruchssicherheit, Datenauthentizität und -integrität sicher zu stellen.

Diese Mechanismen können abhängig sein von der Hardware, in der die Software eingebettet ist, oder an die zur Verfügung stehenden Klassenbibliotheken beziehungsweise API's anzupassen sein. Die dabei in generierten Code einzusetzenden Codestücke sind vom Codegenerator nicht vorauszusehen, da beispielsweise durch neue Plattformen, neue Steuergeräte oder neue Versionen von Klassenbibliotheken ein stetiger und schneller Wandel stattfinden kann. Deshalb ist es wesentlich, dass die Codegenerierung flexibel an die jeweiligen Rahmenbedingungen angepasst werden kann. Dazu gibt es zwei wesentliche Ansätze:

Generierung für abstrakte Schnittstellen, wie in Abbildung 4.4 illustriert, und die
Parametrisierung der Codegenerierung, wie in Abbildung 4.5 gezeigt.

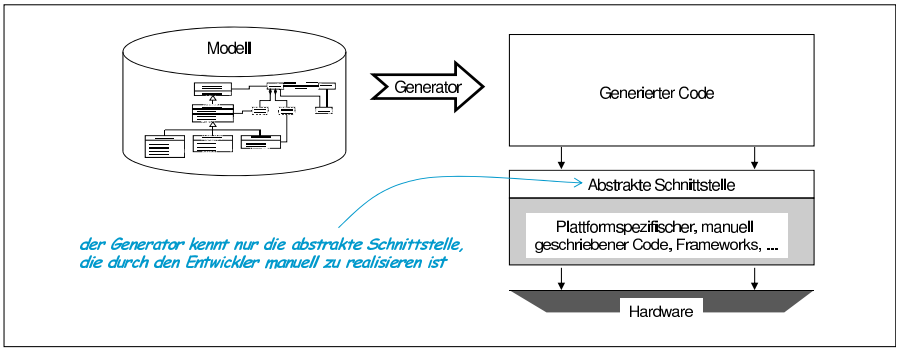


Abbildung 4.4. Generierung von Code gegen eine abstrakte Schnittstelle

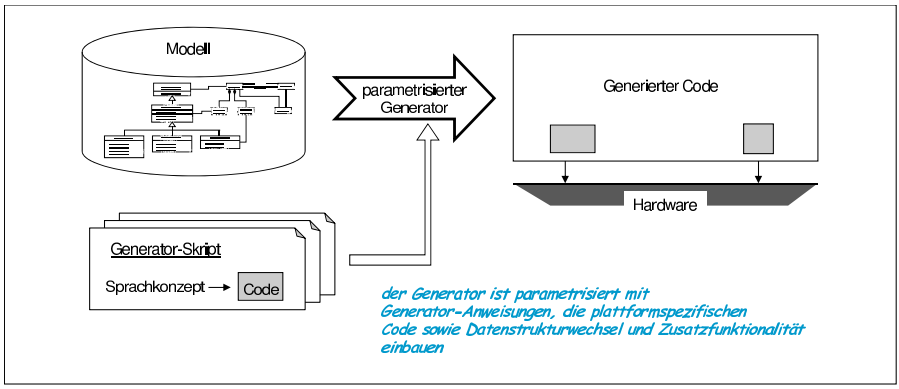


Abbildung 4.5. Parametrisierte Codegenerierung

Für die Trennung von plattformspezifischem und hardwareunabhängigem Code ist die Bildung einer abstrakten Schnittstelle und damit die Schichtentrennung ein ideales Werkzeug, das die Portabilität von Software verbessert. Viele der Java-API's sind genau für diesen Zweck definiert und zum Standard erhoben worden. In [SD00] wurde diese strikte Trennung von Code in Anwendungscode („A-Code") und plattformspezifischen, technischen Code („T-Code") detaillierter untersucht sowie notwendige Mischformen identifiziert. Eines der Ergebnisse dieser durch die Praxis untermauerten Untersuchungen ist dabei, dass eine standardisierte „T-Architektur", also der technische Code für die Speicherung, Anzeige, Fehlerbearbeitung und ähnliche standardisierbare technische Funktionalitäten ein hohes Potential zur Wiederverwendung hat. Dieses Potential kann bei der Codegenerierung durch die flexible Kombination von T-Architektur-Anteilen mit den vom Entwickler vorgegebenen Applikations(A)-Modellen ausgeschöpft werden.

Die strikte Trennung der beiden Codearten kann aber wie immer bei der Einführung von Schichten und Adaptern zu Ineffizienzen führen. Beispielsweise basiert unser Auktionssystem auf asynchroner Kommunikation von Nachrichten. Wenn die abstrakte Schnittstelle aber nur einen RPC-Mechanismus zur Verfügung stellt, so muss unter anderem die Pufferung der Nachrichten selbst codiert werden. Auf unterster Ebene wird aber wieder asynchron über das Internet kommuniziert, wo Puffermechanismen bereits eingebaut sind. Eine Effizienzsteigerung um einen deutlichen Faktor kann zum Beispiel durch Aufgabe der konzeptuell vorhandenen Schichtenbildung und durch ein „Verweben" höherer und niederer Schichten erreicht werden.[11]

[11] Tatsächlich entstehen diese Schichten auch hier, jedoch ist die Schichtenarchitektur nicht horizontal (also Schicht für Schicht) entwickelt, sondern vertikal, so dass die Bedürfnisse höherer Schichten direkt beim Entwurf niederer Schichten berücksichtigt werden konnten. Der Wechsel von RPC zu der selbst realisierten asynchronen

Alternativ kann die angebotene abstrakte Schnittstelle auch breit angelegt sein und im Beispiel sowohl synchronen RPC als auch asynchrone Kommunikation anbieten. Das führt aber zu erheblichem Mehraufwand bei der Realisierung und Weiterentwicklung und zahlt sich nur aus, wenn ausreichend oft eine Wiederverwendung in anderen Projekten stattfindet.

Wird darüber hinaus angenommen, dass die Generierung des Zielcodes korrekt ist und auf eine manuelle Nachbearbeitung oder Inspektion verzichtet, so ist die Einhaltung architektureller Guidlines wie etwa die Schichtenbildung im generierten Code nicht sehr relevant. Stattdessen kann bei der Generierung mehr auf Effizienz geachtet werden und ähnlich zu Optimierungstechniken der Compiler ein Verweben des plattformunabhängigen und -spezifischen Codes erfolgen. Dadurch entsteht nach [SD00] ein sehr schwer wartbarer AT-Code, der sowohl Anwendungs- als auch technisches Wissen beinhaltet. Auch deshalb ist es wesentlich, dass der generierte Code nicht manuell weiterbearbeitet wird, sondern nur die nach A- und T-Gesichtspunkten getrennten Ausgangsmodelle und die Generatorskripte.

In der Praxis ist davon auszugehen, dass eine Mischform aus beiden Generierungsmechanismen zu den besten Ergebnissen führen wird. Darüber hinaus wird ein System eine weitere Komponente besitzen, die eine Laufzeitumgebung für bestimmte Funktionalitäten zur Verfügung stellt, die weder in Java-Klassenbibliotheken noch in Java-Sprachkonzepte abgebildet werden können. Dazu gehören zum Beispiel erweiterte Funktionalitäten zur Behandlung der in OCL verfügbaren Mengen und Listen ebenso wie die Bearbeitung von explizit im Code abgelegten Zustandsmodellen. Abbildung 4.6 beschreibt daher die prinzipielle Struktur eines Codegenerators.

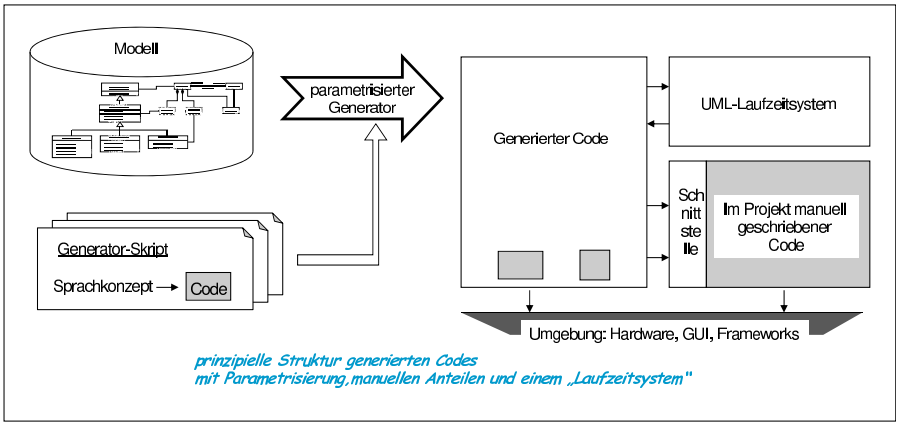


Abbildung 4.6. Struktur eines Codegenerators

Kommunikationsform wurde übrigens mit Refactoring-Techniken [Fow99] effizient umgesetzt.

Der unter anderem in [BBWL01, RFBLO01] geprägte Begriff der „UML Virtual Machine“ entspricht dabei dem rechten Teil des Bildes 4.6, bestehend aus dem UML-Laufzeitsystem und einer plattformspezifischen Implementierung der festgelegten Schnittstellen. Angelehnt an die „Java Virtual Machine“, dem Interpreter des Java-Bytecodes, entspricht der Codegenerator dem Java-Compiler. Die „UML Virtual Machine“ stellt eine Art operationeller Semantik des ausführbaren Teils der UML/P dar.

4.2.2 Funktionalität und Flexibilität

Die im letzten Abschnitt angesprochene Parametrisierung der Codegenerierung kann nicht nur zur Anpassung an plattformspezifische Merkmale verwendet werden, sondern auch dafür, den erzeugten Code um zusätzliche Funktionalität zu erweitern. Im Prinzip sind der dabei entstehenden Flexibilität kaum Grenzen gesetzt. Nachfolgend wird dies an dem einfachen und weitgehend bekannten Beispiel der Codegenerierung für Attribute im Klassendiagramm diskutiert.

Ein in einer Klasse des Klassendiagramms definiertes Attribut besitzt die in Abbildung 4.7 demonstrierte „natürliche“ Umsetzung als Attribut im generierten Java-Code. Mit Ausnahme der Merkmale für abgeleitete und für nur lesbare Attribute (`/` und `readonly`) können alle Merkmale, Typen und initialen Zuweisungen an das Attribut direkt umgesetzt werden. Diese direkte Umsetzung birgt jedoch einige Nachteile, wie zum Beispiel den nicht gekapselten und nicht synchronisierten Zugriff durch andere Objekte.

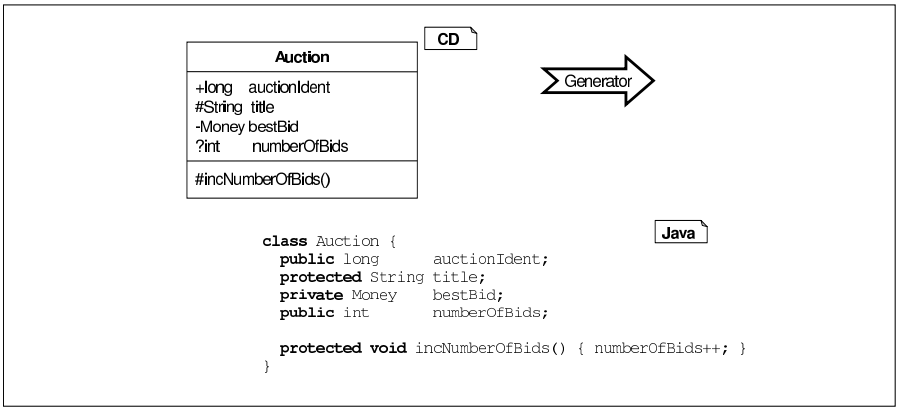


Abbildung 4.7. Direkte Umsetzung von Attributen

Deshalb ist es heute nicht üblich, Attribute aus Analyse- und Entwurfsmodellen direkt in Attribute der Implementierung umzusetzen, sondern stattdessen eine Infrastruktur in Form von so genannten `get`- und

`set`-Methoden zur Verfügung zu stellen. Abbildung 4.8 zeigt die so entstehende Codestruktur. Dabei wird oft jedem Attributnamen ein geeigneter Präfix (hier zum Beispiel der Unterstrich „_“) vorangestellt. Die Verwendung von Zugriffsfunktionen erhöht die Flexibilität. Sie erlaubt zum Beispiel die möglicherweise notwendige Synchronisation von Threads oder die Realisierung des Zugriffsrechts `readonly` durch zwei `get/set`-Methoden mit unterschiedlichen Sichtbarkeitsangaben.

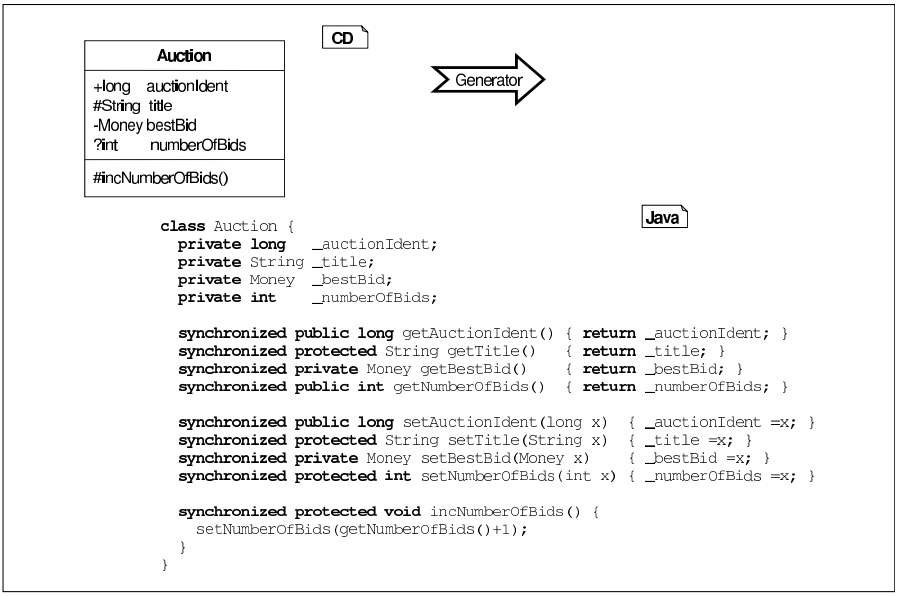


Abbildung 4.8. Umsetzung von Attributen mittels Zugriffsfunktionen

Die in den beiden Abbildungen 4.7 und 4.8 skizzierten Umsetzungen sind heute relativ verbreitet, aber keineswegs die einzigen. Es gibt weitere Varianten, die zum Beispiel persistente Attribute, eine Ablage der Attribute in Enterprise JavaBeans [BR11], Propagierung von Attributänderungen und dergleichen mehr erlauben. Um die verschiedenen und auch nicht generell vorhersehbaren Varianten der Codegenerierung dennoch flexibel zu ermöglichen, ist es grundsätzlich notwendig, die Umsetzung von Konzepten der UML/P stark zu parametrisieren und technologiespezifisch ergänzbar zu halten.

In gewisser Weise sind für die Konzepte der UML/P „API's“[12] identifizierbar, die jeweils umzusetzen sind. Für das Konzept „Attribut“ der Klassendiagramme lässt sich beispielsweise mindestens folgendes API identifizieren:

[12] API im Sinne eines *abstract programming interface*.

- Setzen eines Attributs,
- Auslesen eines Attributs und
- Initialisierung eines Attributs mit einem Defaultwert.

Erweiterungen dieses API's können zum Beispiel für

- Serialisierung,
- Laden aus und Speichern in einer Datenbank,
- Bildschirmausgabe und Einlesen aus einer Bildschirmmaske

oder für typspezifische Funktionalitäten definiert werden. Dazu gehören zum Beispiel die Inkrementierung von Zahlenwerten, das Anhängen von Strings (analog dem Java-Operator `+=`) oder die Behandlung einzelner Elemente in Containerstrukturen.

Die für eine Codegenerierung wünschenswerte Flexibilität besteht also nicht nur in der Form der Umsetzung von UML/P-Konzepten, sondern auch in der damit angebotenen Funktionalität. Die angebotene Funktionalität muss nicht nur generiert werden, sondern auch in einer Form zur Verfügung stehen, die es dem Entwickler erlaubt, an anderer Stelle darauf zuzugreifen. Dabei gibt es zwei generelle Verfahren:

1. Die Umsetzung des API's wird offen gelegt, indem zum Beispiel aus dem Namen und dem Typ eines Attributs eindeutig die Namen und Signaturen der jeweils verwendbaren Funktionen abgeleitet werden können.
   Ist also beispielsweise das Attribut `title` im Klassendiagramm definiert, so kann in Java mit `getTitle()` und `setTitle(...)` darauf zugegriffen werden.
2. Es wird das API selbst offengelegt, die Umsetzung aber bleibt verborgen. Der manuell geschriebene Java-Code nutzt daher direkt das API und muss bei der Codegenerierung ebenfalls transformiert werden.
   Im Beispiel wird dann auch im Java-Code das Attribut `title` benutzt. Dies wird je nach Anwendungsform (lesend oder schreibend) bei der Codegenerierung durch eine `get`- oder `set`-Methode ersetzt.

Während der erste Ansatz zu einfacheren Codegeneratoren führt und keine Behandlung des Java-Codes erfordert, ist beim zweiten Ansatz mehr Flexibilität und Codierungssicherheit gegeben. Durch das Verbergen der tatsächlichen Implementierung kann diese relativ einfach ersetzt oder ergänzt werden. Außerdem ist die Verwendung der API abstrakter und führt zu kompakterem Code. Jedoch wird es in diesem Ansatz notwendig, bei der Codegenerierung auch direkt in Java formulierte Codeteile zu transformieren.

Wie das relativ einfache Beispiel zur Realisierung von Attributen zeigt, lassen sich bereits daran viele der auftretenden Effekte studieren. Deshalb wird im folgenden Abschnitt 4.4 zunächst eine lesbare Form der Darstellung von Codegenerierungen beschrieben und deren Einsatzfähigkeit anhand der Transformation von Attributen demonstriert.

Die bei der Umsetzung von UML/P-Konzepten in die Implementierung zur Verfügung stehenden Formen der Codegenerierung haben unter Umständen Auswirkungen auf die Semantik der Konzepte. Dies ist nicht unkritisch, aber auch eine Chance für Anwender der UML, die *semantischen Freiheitsgrade*, oft auch als *„variation points"* bezeichnet, zu nutzen, um projektspezifische oder zusätzliche Funktionalitäten und Fähigkeiten zu integrieren. Diese Freiheitsgrade präzise zu beschreiben ist innerhalb der UML nicht möglich, einerseits weil die UML selbst keine Mechanismen für ihre Semantikdefinition zur Verfügung stellt, andererseits weil diese Freiheitsgrade keine allgemeine Gültigkeit besitzen, sondern abhängig von potentiellen Zielplattformen und gewünschten Funktionalitäten sind.[13] Es bleibt daher in der Praxis oft nur die Möglichkeit, jeweils einzelne Formen der Codegenerierung und die damit intendierte semantische Interpretation in konstruktiver Form anzugeben. Eine umfassende Beschreibung der Menge möglicher Varianten in ihrer gesamten Bandbreite erscheint nicht möglich. In [Grö10] wurden Feature-Diagramme eingesetzt, um Variationspunkte in Sprachen zu definieren und zum Beispiel in [GRR10, GR10] an mehreren Diagrammen angewendet.

Die Trennung der Funktionalität und der plattformabhängigen Codeteile wurden im Bereich des Aspect-Oriented-Programming (AOP) [KLM+97, LOO01] beziehungsweise der damit eng verwandten generativen Programmierung [CE00] bereits eingehend diskutiert. Dabei werden Techniken vorgestellt, die eine noch weitergehende Trennung einzelner Programmaspekte erlauben. Spezielle Verfahren (so genanntes „Weaving") erlauben die Kombination zunächst unabhängig voneinander formulierter und meist nur über eine abstrakte Programmierschnittstelle verbundener Codeteile. Diese Vorgehensweise wird in eingeschränkter Form auch bei der hier diskutierten Codegenerierung verwendet.

Auch die in [Pre97, Pre00, KPR97] diskutierte Komposition von Klassen aus *Features* und deren Interaktionen kann durch einen generativen Ansatz realisiert werden. Welche Klasse bei der Generierung welche (zusätzlichen) Features erhält, kann durch Stereotypen und Merkmale in geeigneter Form gesteuert werden.

4.2.3 Steuerung der Codegenerierung

Um die bereits mehrfach erwähnte, notwendige Flexibilität in der Codegenerierung in der vollen Bandbreite zu nutzen, muss die Übersetzung sinnvoll gesteuert werden können. So ist es oft sinnvoll, verschiedene Ausprägungen

[13] Es ist allenfalls möglich, durch die Verwendung von Stereotypen auf eine spezielle Bedeutung eines Konstrukts hinzuweisen. Die präzise Definition des Stereotyps und der intendierten Semantik ist in der UML nicht möglich. Stattdessen kann aber eine informelle Beschreibung, wie die in Abschnitt 2.17, Band 1 dargestellte, verwendet werden.

desselben Konzepts innerhalb eines Projekts, unterschiedlich zu realisieren. Die bereits mehrfach diskutierten Möglichkeiten zur Umsetzung von Attributen können zum Beispiel abhängig sein von

- der Klasse, die das Attribut beinhaltet, weil diese Klasse Aufgaben wie Datenhaltung oder Applikationssteuerung haben kann oder als Schnittstelle zu anderen Systemteilen wirkt und daher zu synchronisieren ist,
- der Aufgabe des Attributs innerhalb der Klasse, weil beispielsweise die Klasse persistent ist, das Attribut aber berechnet werden kann oder nur temporäre Daten beinhaltet oder
- dem Typ des Attributs, weil dieser beispielsweise in UML/P, nicht aber in Java existiert.

Diese statisch, also zur Zeit der Codegenerierung festgelegten Abhängigkeiten können durch dynamische Abhängigkeiten ergänzt werden, wenn zum Beispiel ein Flag benutzt wird, um festzulegen, ob ein Objekt persistent sein soll. Projektspezifische Fälle wie diese sollten typischerweise nicht mehr durch eine vorgegebene Semantikdefinition, sondern durch selbstdefinierten Code realisiert werden, der durch den mit Templates parametrisierten Codegenerator systematisch hinzugefügt wird.

Die Steuerung der Implementierungsform jedes UML/P-Konzepts kann grundsätzlich durch Stereotypen und Merkmale erfolgen. Jedoch ist die ausschließliche Verwendung von Stereotypen und Merkmalen in der Praxis nicht ausreichend. Stattdessen reduziert sich die Markierung der UML-Diagramme im Wesentlichen auf den Stereotyp-Namen, optional ergänzt um zusätzliche Parameterwerte. Die Umsetzung eines mit einem Stereotyp markierten UML/P-Elements in eine Implementierung wird durch zusätzliche *Templates* oder *Skripte* vorgenommen, die durch den Codegenerator ausgeführt werden und diesen, wie in Abbildung 4.6 skizziert, sehr flexibel parametrisieren.

4.3 Semantik der Codegenerierung

Eine Codegenerierung ist im Prinzip eine Transformation eines Modells einer Sprache in eine andere Sprache. Auch eine Semantikdefinition ist im Wesentlichen eine Abbildung einer als unbekannt betrachteten Sprache (hier also UML/P) in eine als bekannt und als verstanden angesehene *Zielsprache*. Ist die Zielsprache darüber hinaus *formal* und die Abbildung präzise formuliert, so wird auch von einer *formalen Semantik* gesprochen. In diesem Sinn kann eine durch ein Programm implementierte Abbildung der UML/P in die Programmiersprache Java selbst als eine formale Semantik verstanden werden [HR04]. Dabei gibt es allerdings mehrere Punkte, die bei dieser Argumentation zu beachten sind:

1. Es ist für das Verständnis einer Sprache wie UML/P hilfreich, mehr als nur eine Bedeutungserklärung (Semantik) zur Verfügung zu haben. Durch Verwendung mehrerer Herangehensweisen zur Definition einer Semantik einer Sprache werden unterschiedliche Probleme erkannt und können so in die Sprachdefinition und deren Benutzung (Analyse, Tests, Refactoring) zurückfließen.
2. Der generierte Code ist typischerweise nicht unbedingt gut lesbar, denn er enthält im Allgemeinen eine Vielzahl von technologie- oder frameworkspezifischen Details, die zudem nicht zur eigentlichen Semantik des Modells beitragen. Es ist daher nicht allgemein zumutbar, zum Verständnis der Bedeutung einer Modellierungssprache den generierten Code inspizieren zu müssen. Allerdings gibt es eine Reihe von Sprachbeschreibungen, die darauf beruhen, das Prinzip der Codegenerierung allgemein zu diskutieren. Sie sprechen damit ein breites Publikum an, da heute Programmiersprachen wie Java die am verbreitetsten „formalen Sprachen" sind.
3. Bestimmte Aspekte einer Sprache können bei der Umsetzung in ablauffähigen Code nicht oder nur mit großem Aufwand umgesetzt werden. Dazu gehören zum Beispiel Konzepte, die *Unterspezifikation* erlauben und die in der UML/P an mehreren Stellen eingebaut sind. So können initialisierende Werte für Attribute fehlen oder im Statechart mehrere alternative Transitionen gleichzeitig schaltbereit sein. Der dabei entstehende Nichtdeterminismus wird bei einer Codegenerierung im Normalfall durch die Auswahl einer höher priorisierten Transition aufgelöst (siehe Abschnitt 5.4.4, Band 1). Das durch den generierten Code beschriebene System ist daher im Allgemeinen nicht identisch zum Ausgangsmodell, sondern stellt eine von mehreren möglichen *Spezialisierungen* dar. Weil eine Programmiersprache per se ausführbar ist, ist die Abbildung von Unterspezifikation und damit eine vollständige Semantikdefinition der UML/P zum Beispiel nach Java prinzipiell nicht möglich.
4. Die UML/P ist teilweise auf Ausführbarkeit ausgelegt, erlaubt jedoch an vielen Stellen die Verwendung von nicht ausführbaren Konzepten. Dazu gehören neben den bereits erwähnten Möglichkeiten, Modellinformationen wegzulassen, unter anderem die Spezifikation von Bedingungen mit unendlichen Quantoren. Auch die nachfolgend noch genauer diskutierte Frage der Umsetzung von OCL-Nachbedingungen gehört zu diesem Problemkreis. Daher ist eine vollständige Abbildung von UML/P nach Java nicht möglich.

Als Alternative zu diesen sehr impliziten Semantikdefinitionen lassen sich Techniken der formalen Methoden einsetzen, um eine formale Semantik für die Quellsprache, unabhängig von irgend einer Form der Codegenerierung, zu definieren. Typischerweise sind solche Semantikdefinitionen Abbildungen die eine UML-Variante in eine geeignete Zielsprache transformieren. Als Zielsprachen werden dabei mathematisch formale Kalküle ver-

wendet und die Abbildungen in kompakter Form definiert, so dass sie einer Analyse leichter zugänglich werden.[14] Wie in [HR00] und [Rum98] argumentiert, kann die Existenz zweier Abbildungen für eine Quellsprache verwendet werden, um das Zutrauen in die Korrektheit beider Abbildungen und damit insbesondere in die Codegenerierung zu erhöhen.

Ist die Codegenerierung in der hier vorgeschlagenen Form parametrisiert und haben die benutzten Skripte Einfluss auf Verhalten und Struktur des generierten Codes, so kann dies auf zwei Arten in eine formale Semantikdefinition einbezogen werden. Abbildung 4.9 formalisiert eine Variante zur Semantikdefinition, in der die Semantikabbildung unabhängig vom benutzten Skript ist.

Die in Abbildung 4.9 dargestellte Formalisierung nutzt die mengenwertige Semantikabbildung zum Beispiel auf ein Systemmodell [BCGR09b], um damit die Variabilität des parametrisierten Codegenerators darzustellen. Die Formalisierung hängt sehr stark von den dadurch *beobachteten* Aspekten einer Sprache ab. Wird zum Beispiel nur das extern sichtbare Verhalten formalisiert, so sind in Bezug auf Umsetzung von Attributen, Assoziationen und anderen Strukturelementen Freiheiten gegeben. Tatsächlich ist es für eine so umfangreiche Sprache wie die UML kaum praktikabel, eine vollständige Formalisierung vorzunehmen, obwohl dies in [Öve00] bemerkenswert vollständig, aber nicht sehr elegant gelungen ist. Stattdessen ist es sinnvoll, einzelne, kritische Aspekte genauer zu beleuchten und damit Rückkopplung in den Standardisierungsprozess zu geben. In einer Reihe von Publikationen wurden auch die prinzipiellen Vorteile und Probleme einer Standardisierung diskutiert [BHH+97, FELR98b, FELR98a].

Eine alternative Sichtweise zu der in Abbildung 4.9 dargestellten Form ist die Einbeziehung der Skriptsprache S in die Semantikdefinition. Quellsprache und Skriptsprache stellen dann in gewisser Weise die gemeinsame „Programmiersprache“ dar. Eine Semantikdefinition kann dies in Form einer Funktion $Sem_p : UML \times S \rightarrow Z$ widerspiegeln, die die Auswahl genau eines Elements der Zielsprache Z vornimmt.

4.4 Flexible Parametrisierung eines Codegenerators

In diesem Abschnitt wird zunächst die Frage nach einer geeigneten Darstellungsform für Generatoren und allgemeinen Transformationen diskutiert und dann eine kompakte Repräsentation der Effekte solcher Skripte eingeführt. Im Kapitel 5 werden die Eigenschaften dieser Skripte anhand einer exemplarischen Umsetzung der UML/P-Konstrukte demonstriert. Die hier

[14] Eine sich davon dezidiert unterschiedene Form, zum Beispiel eine axiomatische Semantikdefinition für die UML oder einzelnen Teilen, ist bisher zum Beispiel in [EHHS00] und [EH00b] zu finden und basiert auf einem transformationellen Ansatz dem Graphgrammatiken zugrunde liegen.

Zur **Formalisierung einer Sprache und der Codegenerierung** werden folgende Definitionen benötigt:

- die Quellsprache (UML/P) als eine Menge UML von syntaktisch wohlgeformten Ausdrücken,
- eine geeignete formale Zielsprache mit dem Sprachschatz Z,
- die Skriptsprache des Codegenerators mit dem Sprachschatz S und
- die Menge J aller Java-Programme.

Ein *Codegenerator* ist eine unter Umständen partielle Abbildung $Gen : UML \rightarrow J$. Eine *formale Semantik* ist demgegenüber eine Abbildung $Sem : UML \rightarrow \mathbb{P}(Z)$. Damit wird einem einzelnen typischerweise unterspezifizierten und abstrakten Modell aus der Quellsprache eine Menge von möglichen Implementierungen zugewiesen, die mit eben diesem Modell gemeint sind. Dies stellt eine Form der *losen Semantik* dar.
Für einen Vergleich beider Abbildungen Sem und Gen ist eine Semantik für Java-Programme in der Form $Sem_{Java} : J \rightarrow Z$ notwendig. Dann muss für jedes UML-Dokument $u \in UML$, für das Code generiert werden kann, gelten:

$$\forall u \in UML : Sem_{Java}(Gen(u)) \in Sem(u)$$

Das heißt, im Allgemeinen wählt der Codegenerator eine von mehreren möglichen Implementierungen aus, indem er etwa offene Aspekte durch Defaults ausfüllt. Nur wenn $Sem(u)$ ein einziges Element darstellt, war die Spezifikation offensichtlich vollständig und eindeutig.
Ein *parametrisierter Codegenerator* wird um die Parameter, also die Skriptsprache S, erweitert: $Gen_p : UML \times S \rightarrow J$. Es muss nun gelten:

$$\forall u \in UML, s \in S : Sem_{Java}(Gen_p(u, s)) \in Sem(u)$$

Das heißt, im Rahmen der Vorgabe durch $Sem(u)$ darf das Skript s eine mögliche Implementierung für u auswählen.

Abbildung 4.9. Semantik des parametrisierten Codegenerators

diskutierten Konzepte stellen for allem Bezug zu einem Codegenerator her, sind aber auf andere Formen von Analysewerkzeugen und Transformatoren ebenfalls anwendbar. Zum Beispiel können damit auch Datenstrukturwechsel oder Refactoring-Techniken beschrieben werden.

4.4.1 Implementierung von Werkzeugen

In Abschnitt 4.2.3 wird die Verwendung von Skripten bzw. Templates zur flexiblen Parametrisierung eines Codegenerators, aber auch für Analyse- und Testwerkzeuge diskutiert. Tatsächlich erfordert eine plattformspezifische und eine auf verschiedene Implementierungsmöglichkeiten ausgerichtete Codegenerierung einen flexiblen Mechanismus zur Steuerung der Codegenerierung. Die Generierung kann im Prinzip durch Stereotypen und Merkmale gesteuert werden. Die Details des zu erzeugenden Codes können mit

Stereotypen aber nicht oder zumindest nicht komfortabel festgehalten werden.

Im Allgemeinen können die zu erzeugenden Codestrukturen sehr komplex sein und unterschiedlichste Formen haben. Meist sind Symboltabellen notwendig und Nebenbedingungen zu prüfen, die die Anwendbarkeit einer Codegenerierung oder mögliche Optimierungen beschreiben. Aufgrund der erforderlichen Flexibilität kommt nur eine in ihrer Beschreibungsmächtigkeit (weitgehend) vollständige Programmiersprache in Betracht, die eine kompakte Formulierung von Bedingungen und Transformationen erlaubt. In dieser Sprache formulierte Transformationen werden vom Codegenerator zur Ausführung gebracht, um den Implementierungscode zu erzeugen. Sie sind selbst zur Laufzeit nicht vorhanden. Abbildung 4.10 beschreibt die daraus resultierende innere Struktur eines Generators.

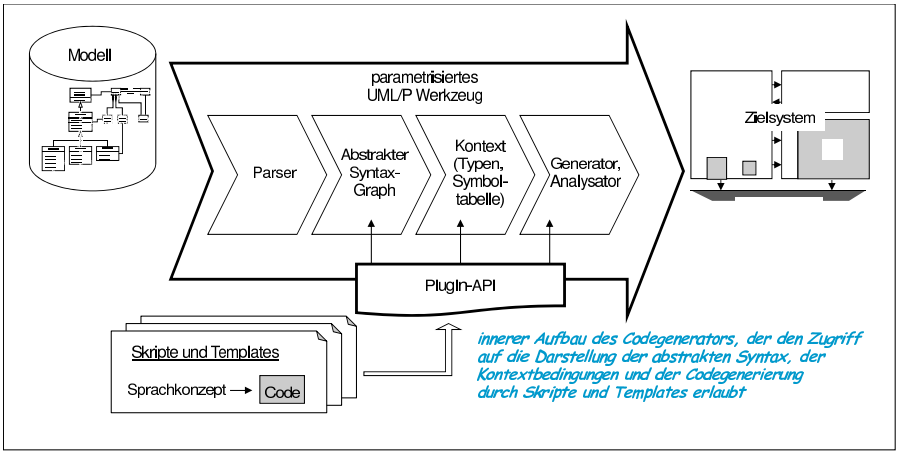


Abbildung 4.10. Innere Struktur eines Codegenerators

Für Skript- oder Templatesprachen wurden in der Werkzeugentwicklung unterschiedliche Vorschläge gemacht. So sind interpretierte Derivate der Sprache C ebenso in Verwendung wie Visual Basic, Skriptsprachen wie Pearl oder Tcl/Tk. Funktionale Programmiersprachen wie ML [Pau94, MTHM97] bieten darüber hinaus eine sehr kompakte Form, solche Werkzeuge zu parametrisieren. Zum Beispiel ist das Verifikationswerkzeug Isabelle [NPW02] in ML geschrieben und bietet damit gute, ebenfalls in ML zu formulierende Erweiterungsmöglichkeiten.

Eine weitere Alternative ist die mittlerweise populäre XML-Technologie [McL06, W3C00] und die Verwendung von XML/XSLT-basierten Werkzeugen zur Transformation in den Zielcode. Allerdings besitzt XML durch die explizite Einbettung der Tags (Nichtterminale) einerseits ein sehr schlechtes Verhältnis zwischen Struktur- und Nutzinformation und andererseits sind

die heutigen Parse- und Transformationswerkzeuge für die XML noch nicht so leistungsfähig, wie die im Bereich des Compilerbau längst bekannten `Yacc/Lex` und deren Derivate.

Mit ihrer Vielzahl an Bibliotheken, Frameworks und Werkzeugen sowie der Möglichkeit des dynamischen Ladens von Klassenbibliotheken ist auch Java ein guter Kandidat, einerseits, um damit einen Codegenerator zu realisieren und andererseits, um damit die Parametrisierung über einen Plugin-Mechanismus zu ermöglichen. Beispielsweise nutzt das Werkzeug `Poseidon` [BS01a, BBWL01] diesen Mechanismus.

Eine aktive Skriptsprache besitzt jedoch unkomfortable Defizite bei der Komposition von Artefakten der Zielsprache. Deshalb bietet sich zusätzlich die Verwendung eines Makro-Ersetzungsmechanismus an, der Templates[15] verarbeitet und daraus Code erzeugt. Ein Template ist als passive Skriptsprache zu sehen, die Makros beinhaltet, die durch den Ersetzungsalgorithmus zum Beispiel durch echte Namen oder Ausdrücke ersetzt werden. Ein Template kann aktive Elemente beinhalten, um damit Kontrollstrukturen, wie Alternative, Wiederholung oder Einbindung anderer Templates zu ermöglichen, aber auch um Berechnungen in beschränktem Maß durchzuführen. Verwendbar sind hierbei XML auf der Transformationssprache XSLT basierende Werkzeuge oder ein mit Java-Code kombinierter Template-Mechanismus ähnlich der JSP-Seiten [FK00], der zum Beispiel von [SvVB02] vorgestellt wird.

Aus der flexiblen Programmierbarkeit des Generators folgt, dass Entwickler nicht nur in der Entwicklungsprogrammiersprache, sondern in eingeschränkter Form auch in der Skriptprogrammiersprache entwickeln müssen. Aufgrund der regelmäßig notwendigen technologieabhängigen Anpassungen muss dies parallel zur eigentlichen Entwicklung geschehen und daher zumeist im selben Projekt stattfinden. Es wäre daher für das Erlernen von Vorteil, wenn sich Skript- und Zielprogrammiersprache ähnlich sind.

Im Generator-Framework MontiCore [KRV10, Kra10, KRV08, GKR+08] wird zum Beispiel die templatebasierte Java Template Engine FreeMarker [Dib01] eingesetzt, um aus UML-Modellen Code zu generieren. Die Trennung zwischen verarbeitendem Frontend (Parser und Analyse der Kontextbedingungen) sowie dem generierenden Backend erlaubt eine flexible Anpassung des generierten Codes. Dies zeigt sich insbesondere bei dem darauf basierten Codegenerator zur UML/P [Sch12]. Dabei treten die FreeMarker-Skriptsprache sowie die Zielsprache in gemischter Form auf. Um die Templates und damit die Zielstruktur des Codes nicht übersichtlich zu managen wird eine Strukturierung der Templates entlang der Zielstruktur des Codes und gleichzeitiger Unterstützung durch MontiCore-Basisfunktionen vorgeschlagen. Das in Java realisierte Generator-Framework MontiCore und damit

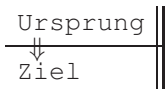
[15] Der englische Begriff „Template“ hat sich in diesem Kontext eingebürgert, so dass er auch in diesem Buch gegenüber der deutschen Version „Codeschablone“ bevorzugt wird.

auch der UML/P-Generator [Sch12] kann um eigene Basisfunktionen erweitert werden, im Allgemeinen kommt der Nutzer aber mit der Templatesprache selbst aus.

Ist eine Entscheidung zugunsten einer Skriptsprache und einer Template-Sprache gefallen, so stellt sich die Frage, ob sich die Effekte der Skripte für den Anwender des Generators in kompakter, verständlicher, aber gegebenenfalls informeller Form darstellen lassen, die nicht auf der doch mit vielen Implementierungsdetails behafteten Skriptsprache beruhen. Dieser Wunsch wird noch verständlicher, wenn die XML-basierte sehr verbose Transformationssprache XSLT verwendet werden muss. In diesem Buch soll aber weniger die konkrete Ausformulierung der Codegenerierung, sondern die Konzeption als Transformationen im Vordergrund stehen, weshalb eine abstrakte Darstellung als Transformationen gewählt wird.

4.4.2 Darstellung von Skripttransformationen

Der Effekt eines für die Codegenerierung verwendeten Skripts beruht darauf, UML/P-Konzepte in die Zielprogrammiersprache Java zu transformieren. Da ein solches Skript auch Rand- und Sonderfälle behandelt, Rahmenbedingungen prüft und dazu eine Reihe von Hilfsfunktionen nutzt, ist es zweckmäßig den Effekt eines solchen Skripts in kompakter Form darzustellen, dabei Sonderfälle nur informell zu diskutieren und so für den Anwender verständlich zu machen. Dazu wird die Schablone in Tabelle 4.11 vorgeschlagen, in der neben der eigentlichen Transformation Kontextbedingungen eine allgemeine Beschreibung angegeben und potentielle Alternativen diskutiert werden können. Diese Beschreibung ist weder als vollständig noch als formal anzusehen, obwohl sie einigen nachfolgend beschriebenen Einschränkungen unterliegt. Diese Schablone stellt eher eine abstrakte Illustration für die Beschreibung von Transformationen dar.

Name der Transformation	
Erklärung Transformationsregel	Motivation und Zweck der Transformation.
	Meist gibt es eine primäre Transformationsregel, die in folgender Form dargestellt wird: 

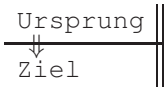
(Fortsetzung auf nächster Seite)

(Fortsetzung von Tabelle 4.11.: Name der Transformation)

	• `Ursprung` und `Ziel` können dabei jeweils textuell oder durch ein Diagramm dargestellt werden. • Der `Ursprung` führt die Elemente der Syntaxklassendiagramme und EBNF-Produktionen ein, die transformiert werden. Der Kontext wird ebenfalls aufgelistet. Dafür werden schablonenartige Diagramme und Textstücke verwendet. • Die kursiven Namen stellen *Schemavariablen* dar, die bei der Anwendung der Transformation mit echten Sprachelementen belegt werden. • Erklärende Texte beschreiben die Arten der Schemavariablen, welche Teile optional sind oder mehrfach auftreten können, etc. • Die Inhalte der Schemavariablen können selbst gewissen Transformationen unterliegen. So kann die Schemavariable *attr* dazu genutzt werden, einen Methodennamen `set`*Attr* zu konstruieren, der aus dem konstanten Teil `set` und der kapitalisierten Form des Attributnamens zusammengesetzt ist.
weitere Transformationen	Aus der Haupttransformation ergeben sich meist zusätzlich notwendige Transformationen, die in analoger Form dargestellt werden. Diese Transformationen beziehen sich zum Beispiel auf Elemente der in Abschnitt 4.2.2 diskutierten API, die ebenfalls zu transformieren sind.
Beachtenswert	Dieser Abschnitt rundet durch zusätzliche Betrachtungen, Hinweise und die Diskussion potentieller Problemstellungen die Beschreibung ab.

Tabelle 4.11.: Name der Transformation

Der verwendete Regelmechanismus ist angelehnt an formale Regelkalküle der Form



Rahmenbedingungen,

die neben `Ursprung` und `Ziel` eine präzise Angabe der Rahmenbedingungen in einer formalen Notation enthalten. Der Ursprung enthält Schemavariablen (Platzhalter, [BBB+85]), die bei der Anwendung der Transformation mit echten Sprachelementen belegt werden. Jede Schemavariable ist einem bestimmten Nichtterminal zugeordnet, ist also damit *typisiert*.

Nachfolgend ein Anwendungsbeispiel für eine Standardtransformation, die die in Abbildung 4.8 exemplarisch gezeigte Codegenerierung vornimmt.[16]

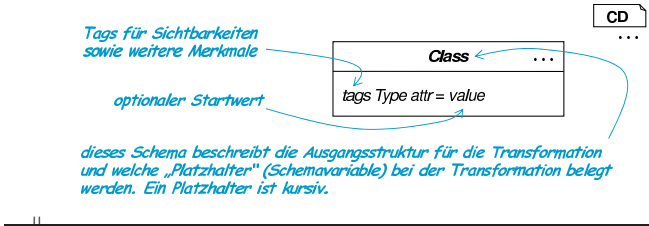
Wie bereits in Abschnitt 4.2.2 besprochen, stellt die hier dargestellte Umsetzung nur eine von mehreren Möglichkeiten dar, die zum Beispiel durch die Verwendung geeigneter Stereotypen und Merkmale am Attribut, der Klasse oder dem Klassendiagramm, aber auch durch Angabe bestimmter Skripte ausgewählt werden können.

Attribut1: Standardtransformation von Attributen

Erklärung

Die Standardform zur Übersetzung von Attributen mit Kapselung und Zugriff durch explizite Zugriffsfunktionen.
Typ- oder projektspezifische Funktionalitäten sind nicht enthalten.

Attributdefinition




```
class Class { ...                                   Java
  private Type attr = value;
  tags' synchronized Type getAttr() {
       return attr;
  }
  tags'' synchronized Type setAttr(Type a) {
       return attr=a;
  }
}
```

- Die optionale Besetzung mit *=value* wird nur verwendet, wenn sie im Diagramm angegeben ist.
- Die in *tags* angegebene Sichtbarkeit wird mit Ausnahme von `readonly` übernommen. `readonly` wird bei `get`*Attr* (also in *tags'*) in `public` umgesetzt und bei *tags''* in `protected`.

(Fortsetzung auf nächster Seite)

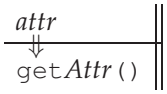
[16] Allerdings unter Verzicht auf die in manchen Codierungsstandards propagierte Verwendung von führenden Unterstrichen für Attribute.

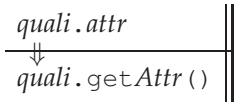
(Fortsetzung von Tabelle 4.12.: Attribut1: Standardtransformation von Attributen)

<table>
<tr><td></td><td>

- `return` ist bei `set`*Attr* sinnvoll, denn die Zuweisung mit = hat denselben Wert.
- Die Kardinalität des übersetzten Attributs ist entweder nicht angegeben und damit „`1`“ oder „`0..1`“.

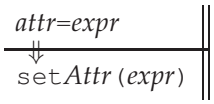
</td></tr>
<tr><td>Attribut-zugriff</td><td>

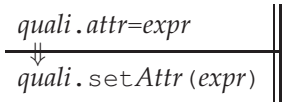




- Typ von Ausdruck *quali* ist konform zur Klasse *Class*.

</td></tr>
<tr><td>Attribut-besetzung</td><td>





- Typ von Ausdruck *quali* ist konform zur Klasse *Class*.
- Typ von Ausdruck *expr* ist konform zum Attributtyp *Type*.

</td></tr>
<tr><td>Beachtens-wert</td><td>

Typspezifische Konstrukte wie *attr*++ können entweder durch zusätzliche Funktionalität effizient umgesetzt oder in `set`*Attr*`(get`*Attr*`()+1)` transformiert werden.
Merkmale und Stereotypen werden in dieser Standardumsetzung nicht berücksichtigt.
In UML ist es möglich, Attributen eine Kardinalität zuzuordnen. Im Fall „`*`“ ist eine Umsetzung ähnlich der Assoziation vorzunehmen.

</td></tr>
</table>

Tabelle 4.12.: *Attribut1:* Standardtransformation von Attributen