COBAMOS

Endnutzerprogrammierung auf Basis nachrichtenbasierter Komponentenvernetzung

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
des Fachbereiches
Physik, Mathematik und Informatik
der
Johannes Gutenberg-Universität Mainz

vorgelegt von
Guido Ralf Töpfer
geboren in Wiesbaden

Mainz, im Juli 2006

Tag der mündlichen Prüfung: 29. September 2006

D 77 (Dissertation an der Johannes Gutenberg-Universität Mainz)

Danksagungen

Eine Dissertation ist ohne die Unterstützung von Menschen im Umfeld – sei es das persönliche, berufliche oder das der Forschung – kaum anzufertigen. An dieser Stelle möchte ich allen Menschen herzlich danken, die mich auf diesem Weg unterstützt haben.

Mein Dank gilt meinem Doktorvater, Herrn Prof. Dr. J.P., für die Überlassung dieses Themas und für seine Diskussionsbereitschaft und die in allen Phasen dieser Dissertation „offene Tür“. Sein kritisches aber stets konstruktives Hinterfragen lieferte mir viele wichtige Anregungen für diese Arbeit.

Dem Zweitgutachter dieser Dissertation, Herrn Privatdozenten Dr. habil. K.B., danke ich für die wertvollen fachlichen Anregungen in vielen interessanten Gesprächen und seine ständige Bereitschaft, sich mit meiner Arbeit auseinanderzusetzen.

Meinem Kollegen Herrn Dr. HJ.S. möchte ich für das freundschaftliche Verhältnis während meiner Zeit am Institut danken. Seine Ratschläge haben mir vieles erleichtert und neben den fachlichen Anregungen möchte ich ihm besonders sein immer offenes Ohr und die unterstützenden und aufbauenden Worte insbesondere in allen Phasen dieser Arbeit danken.

Frau A.D. gilt mein Dank für das Korrekturlesen dieser Arbeit und die unterhaltsamen Diskussionen über die neue und alte deutsche Orthographie.

Ebenso spreche ich allen Mitarbeitern des Instituts für Informatik, die mich bei der Anfertigung dieser Arbeit unterstützt haben, meinen Dank aus.

Ich danke ganz besonders meiner Lebensgefährtin, Frau F.W., und meiner Familie für die Unterstützung und die Kraft, die sie mir während dieser Zeit gegeben haben. Ohne ihr Verständnis wäre diese Arbeit sehr viel mühseliger gewesen.

Zusammenfassung

Die Aufgabenstellung, welche dieser Dissertation zugrunde liegt, lässt sich kurz als die Untersuchung von komponentenbasierten Konzepten zum Einsatz in der Softwareentwicklung durch Endanwender beschreiben. In den letzten 20 bis 30 Jahren hat sich das technische Umfeld, in dem ein Großteil der Arbeitnehmer seine täglichen Aufgaben verrichtet, grundlegend verändert. Der Computer, früher in Form eines Großrechners ausschließlich die Domäne von Spezialisten, ist nun ein selbstverständlicher Bestandteil der täglichen Arbeit. Der Umgang mit Anwendungsprogrammen, die dem Nutzer erlauben in einem gewissen Rahmen neue, eigene Funktionalität zu definieren, ist in vielen Bereichen so selbstverständlich, dass viele dieser Tätigkeiten nicht bewusst als Programmieren wahrgenommen werden. Da diese Nutzer nicht notwendigerweise in der Entwicklung von Software ausgebildet sind, benötigen sie entsprechende Unterstützung bei diesen Tätigkeiten. Dies macht deutlich, welche praktische Relevanz die Untersuchungen in diesem Bereich haben.

Zur Erstellung eines Programmiersystems für Endanwender wird zunächst ein flexibler Anwendungsrahmen entwickelt, welcher sich als Basis zur Erstellung solcher Systeme eignet. In Softwareprojekten sind sich ändernde Anforderungen und daraus resultierende Notwendigkeiten ein wichtiger Aspekt. Dies wird im Entwurf des Frameworks durch Konzepte zur Bereitstellung von wieder verwendbarer Funktionalität durch das Framework und Möglichkeiten zur Anpassung und Erweiterung der vorhandenen Funktionalität berücksichtigt. Hier ist zum einen der Einsatz einer serviceorientierten Architektur innerhalb der Anwendung und zum anderen eine komponentenorientierte Variante des Kommando-Musters zu nennen. Zum anderen wird ein Konzept zur Kapselung von Endnutzerprogrammiermodellen in Komponenten erarbeitet. Dieser Ansatz ermöglicht es, unterschiedliche Modelle als Grundlage der entworfenen Entwicklungsumgebung zu verwenden.

Im weiteren Verlauf der Arbeit wird ein Programmiermodell entworfen und unter Verwendung des zuvor genannten Frameworks implementiert. Damit dieses zur Nutzung durch Endanwender geeignet ist, ist eine Anhebung der zur Beschreibung eines Softwaresystems verwendeten Abstraktionsebene notwendig. Dies wird durch die Verwendung von Komponenten und einem nachrichtenbasierten Kompositionsmechanismus erreicht. Die vorgenommene Realisierung ist dabei noch nicht auf konkrete Anwendungsfamilien bezogen, diese Anpassungen erfolgen in einem weiteren Schritt für zwei unterschiedliche Anwendungsbereiche.

Abstract

The problem on which this thesis is based can shortly be described as the research on component-based concepts to be used in the field of end user software development. In the last 20 to 30 years, the technical workplace environment has fundamentally changed. Computers are no longer the exclusive domain of specialists, today they are a common part of daily work. In many areas, the use of software applications allowing the users in a limited scope to create new functionality, is common. These tasks are usually not seen as software development tasks. Since the users in these scenarios are not necessarily trained software developers, they need congruous assistance. These facts show the importance of research in this field.

We first create a framework as basis for the development of programming environments for end users. Since changing requirements and the resulting needs are an important part in software projects, the framework is created to provide reusable functionality and to be adaptable and extensible. This goal is reached by the use of a service oriented architecture and a component based version of the command pattern. Furthermore, we develop a concept to encapsulate the realizations of models for use in end user programming into components. This approach can be used to configure end user programming environments to suit different needs and forms a base for the development of such environments.

The remainder of this thesis describes the design and realization of a model for end user programming to be used with the framework mentioned afore. The level of abstraction used to describe software applications in this model has to be raised since it is intended for the use by end users. This is done by the use of components to form the building blocks of the applications and the realization of a message-based composition mechanism. The solution presented is not yet directed at the use in a certain problem domain. It is designed in a way that allows for adaptation to concrete domains to be usable in a broader scope. Two examples of such adaptations are presented.

Inhaltsverzeichnis

1 Einleitung

„Auch der längste Marsch beginnt mit dem ersten Schritt.“
Laotse, Chinesischer Philosoph, 6. Jh. v. Chr.

Dieses Kapitel ist der erste Schritt in die vorliegende Dissertation. Es motiviert die im Rahmen dieser Arbeit behandelte Themenstellung und gibt einen kurzen Abriss über die weiteren Kapitel, die Struktur der gesamten Arbeit und das Verhältnis der erarbeiteten Konzepte zueinander.

1.1 Motivation und Aufgabenstellung

Die Aufgabenstellung, die der vorliegenden Arbeit zugrunde liegt, lässt sich kurz als die Untersuchung von komponentenbasierten Konzepten zum Einsatz in der Softwareentwicklung durch Endanwender beschreiben. Dabei ist die Faszination für diese Idee, Anwender ohne Ausbildung in der Softwareentwicklung in die Lage zu versetzen, Software zu entwickeln, ein zentraler Antrieb für diese Dissertation.

Der Umgang mit Computern ist inzwischen ein selbstverständlicher Teil unserer Arbeitswelt geworden und auch im privaten Umfeld ist die Computerisierung allgegenwärtig. Endanwender ohne Ausbildung in der Sofwareentwicklung verwenden Anwendungsprogramme, die ihnen in gewissen Grenzen erlauben, neue Funktionalität zu definieren. Diese Tätigkeiten werden häufig nicht als Programmieren wahrgenommen, ein Umstand der zu gewissen Risiken führen kann (siehe auch Abschnitt 2.3, Gefahren und Nachteile im Zusammenhang mit der Endnutzerprogrammierung). In vielen anderen Szenarien sind Anwender auf die Rolle reiner Konsumenten von Softwareapplikationen festgelegt. Falls keine Anwendung auf dem Markt existiert, die ihren Bedürfnissen entspricht – und dies ist insbesondere für kleine Marktsegmente zu erwarten, wenn diese für Softwarehersteller unrentabel erscheinen – müssen sie sich mit dieser Situation abfinden.

Die Erstellung von Programmiersystemen für Endanwender erfordert zwar im Allgemeinen Einschränkungen hinsichtlich der Applikationen, die sich entwickeln lassen. Dies folgt aus der Notwendigkeit, die Abstraktionsebene anzuheben auf der eine Anwendung beschrieben wird. Dennoch kann diese Möglichkeit in vielen Bereichen vorteilhaft genutzt werden.

Der in dieser Arbeit verfolgte Ansatz verwendet Komponenten zur Anhebung des Abstraktionsniveaus. Er eignet sich prinzipiell dazu, dass ein Softwarehersteller nicht eine unrentable Speziallösung für ein Marktsegment erstellen muss, sondern ein Familie ähnlicher Anwendungen beschreiben kann. Ein Anwender kann in der Folge seine individuelle Applikation aus dieser Familie erstellen. Der Vorteil für den Hersteller besteht darin, dass er – mit etwas höherem Anfangsaufwand – einen interessanten Markt erschließen kann. Der Endnutzer wird in eine emanzipiertere Rolle gegenüber den Softwareherstellern versetzt.

Die vorliegende Arbeit fügt sich dabei in die Reihe einiger Forschungsarbeiten ein, welche insbesondere im Bereich der Sportinformatik am Institut für Informatik der Johannes Gutenberg-Universität durchgeführt worden sind und den Bereich der Softwareentwicklung durch Endanwender berühren. Zu nennen sind hier insbesondere die Projekte CADMoSS (Computer Aided Design and Modelling of Systems in Sport, siehe [Per97]) und CADMOS (Component Aided Design and Modelling of Systems, siehe [Rei00]). In Anlehnung an die Arbeitstitel dieser Projekte wurde für das in dieser Dissertation behandelte die Bezeichnung CoBaMoS – Component Based Modelling of Systems gewählt. Aus Gründen der besseren Lesbarkeit wird im Folgenden die Schreibweise Cobamos verwendet.

Ein wichtiger Aspekt während des gesamten Projektes Cobamos war die Umsetzbarkeit der entworfenen Ansätze. Dieser Anspruch resultiert nicht zuletzt aus der Tatsache, dass dieses Dissertationsvorhaben am Lehrstuhls für angewandte Informatik durchgeführt wurde. Neben konzeptionellen Betrachtungen wird daher in dieser Arbeit immer wieder der Bogen zur möglichen Realisierung der Entwürfe gespannt.

Bei den Überlegungen zur Umsetzung eines möglichen Lösungsszenarios unter Verwendung von Komponenten zeigte sich die Notwendigkeit, ein geeignetes Entwicklungswerkzeug zu erstellen, mit dem sich die erarbeiteten Konzepte auch prototypisch umsetzen ließen. Es wurde jedoch keine Umsetzung eines Prototyps zur ausschließlichen Verwendung in diesem Projekt vorgenommen. Das entworfene Cobamos-Framework soll als Basis zur Realisierung anderer Projekte zur Untersuchung von Konzepten aus dem Bereich der Endnutzerprogrammierung nutzbar sein. Auch zur Realisierung von Simulationsumgebungen kann dieser Ansatz genutzt werden.

In Abb. 1.1 ist die unter Verwendung des Cobamos-Frameworks implementierte Programmierumgebung für Endnutzer gezeigt. Das gezeigte Beispiel soll nur einen kurzen Ausblick auf die weitere Arbeit geben, eine detaillierte Besprechung der Nutzung findet sich in Kapitel 9. Die Anwendung erlaubt die Modellierung von Softwareapplikationen zur Manipulation von Datenströmen aus ganzen Zahlen. Die Applikationen können visuell komponiert werden; in der Mitte der Nutzungsoberfläche ist ein Graph zu sehen, der eine solche Anwendung beschreibt. Zwei Zahlengeneratoren erzeugen Zahlenfolgen, die Evaluatorkomponente verknüpft diese beiden Zahlenströme – im Beispiel werden die Werte addiert – und das Ergebnis wird von einer weiteren Komponente dargestellt.

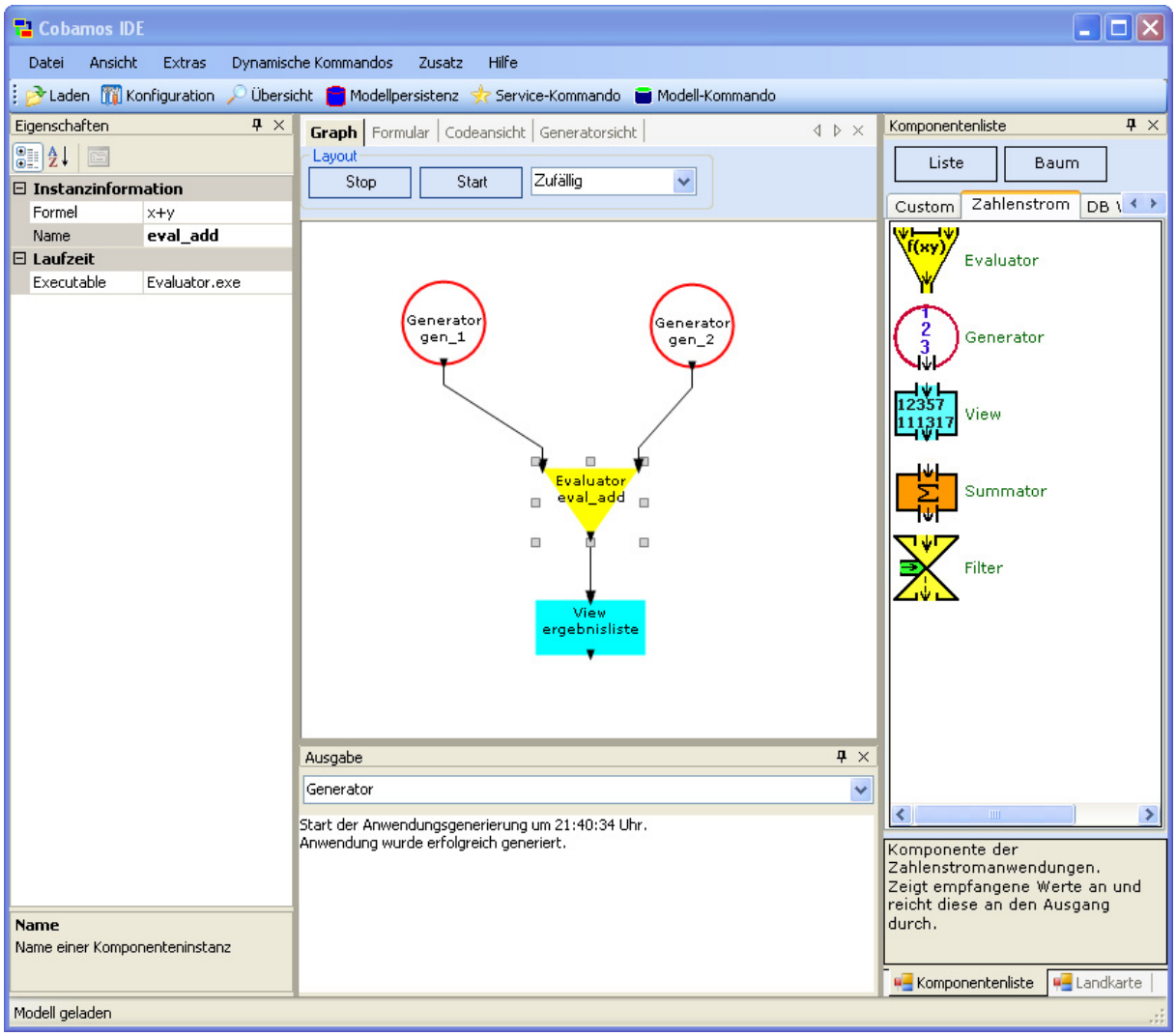


Abb. 1.1: Endnutzerprogrammierumgebung auf Basis des Cobamos-Frameworks

Auf Basis der Ergebnisse dieser Dissertation sind neben der gezeigten Anwendung auch eine Reihe anderer Softwareapplikationen umsetzbar. Viele der Zwischenergebnisse aus diesem Projekt lassen sich auch in abweichender Art und Weise verwenden, an den entsprechenden Stellen wird durch Ausblicke im Text auf solche Möglichkeiten hingewiesen. Im Verlauf der Arbeit ändern sich jedoch die Zielgruppen der erarbeiteten Ansätze, diese reichen von Softwareentwicklern in unterschiedlichen Rollen bis zu Endanwendern. Hier sollen daher zunächst die unterschiedlichen Adressaten und das Verhältnis der Einzelergebnisse zueinander erläutert werden, um den Lesern dieser Arbeit deren Einordnung in den Gesamtkontext zu erleichtern.

In Abb. 1.2 ist der Überblick über die im Rahmen dieser Arbeit entwickelten Konzepte dargestellt. Die Konzepte einer niedrigeren Ebene sind dabei immer als eine Anwendung der Konzepte der darüber liegenden Stufe zu sehen[1]. Die Grundlage der gesamten Entwicklungen ist das bereits erwähnte Cobamos-Framework. Dieses ist als Basis für Entwicklungsumgebungen und vergleichbare Anwendungen gedacht und ist auf eine ausschließliche Nutzung durch Softwareentwickler ausgerichtet.

Das Framework erlaubt die Verwendung unterschiedlicher Basiskonzepte zur Realisierung von konkreten Programmierumgebungen, diese müssen von einem Softwareentwickler (die Rolle wird mit

[1] Die Anordnung von oben nach unten entspricht der Reihenfolge der Behandlung in dieser Arbeit.

Softwareentwickler Entwicklungsumgebung bezeichnet) erstellt werden. Dadurch wird eine konkrete Entwicklungsumgebung geschaffen. In dieser Arbeit wird ein Ansatz zur nachrichtenbasierten Komposition von Komponenten als zentrales Konzept für die Umsetzung einer solchen Entwicklungsumgebung verwendet. Diese Realisierung ist so gestaltet, dass unterschiedliche Familien von Anwendungen damit modelliert werden können. Eine solche Konkretisierung muss ebenfalls durch einen Softwareentwickler vorgenommen werden (Rolle Softwareentwickler Anwendungsfamilie). Beispiele für Anwendungsfamilien sind die im Rahmen dieser Arbeit entwickelten Prototypen zur Realisierung von Workflows im Bereich Datenbankadministration und die bereits erwähnte Anwendungsfamilie zum Umgang mit Datenströmen ganzer Zahlen.

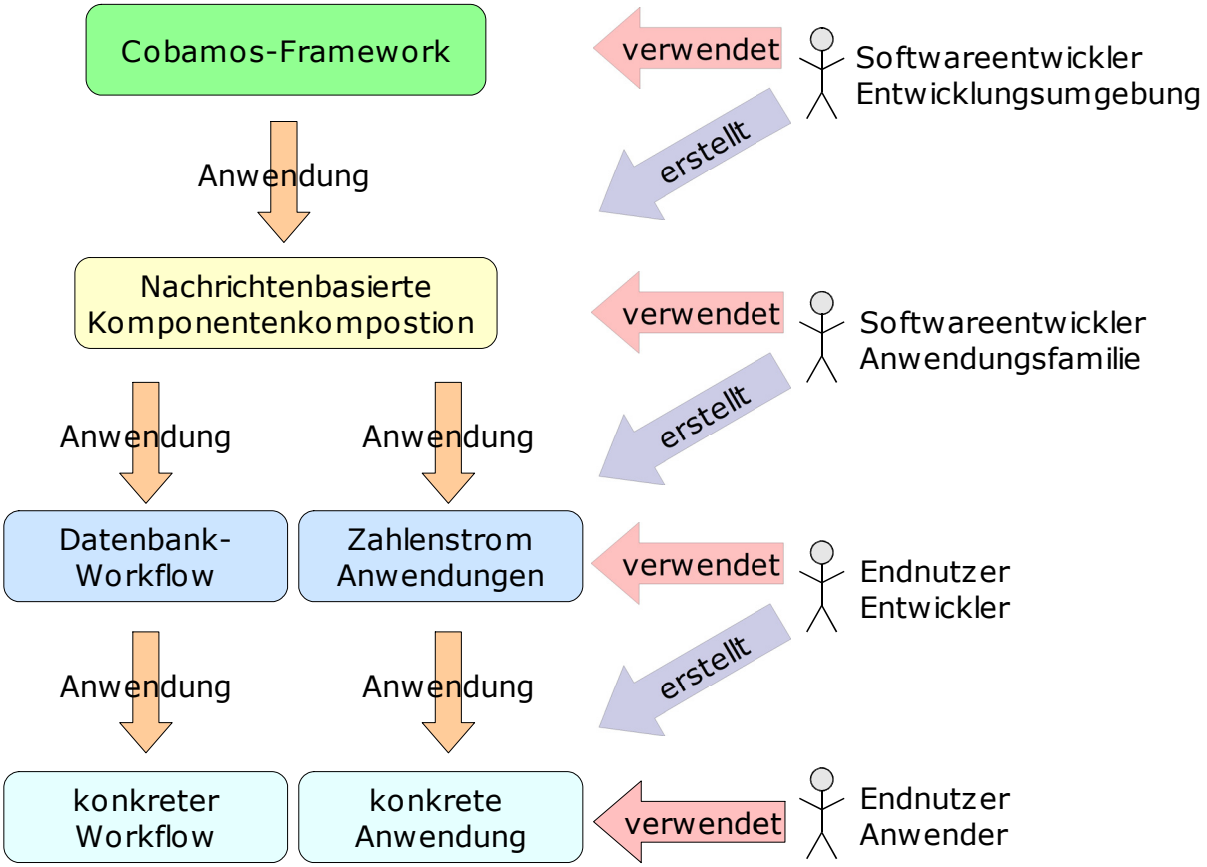


Abb. 1.2: Zusammenhang zwischen entwickelten Lösungsansätzen und Benutzerrollen

Erst diese Ebene der Anwendungsfamilien ist für den Einsatz zur Anwendungsentwicklung durch einen Endnutzer (Rolle Endnutzer Entwickler) gedacht. Die von diesem erstellten Anwendungen sind in der untersten Ebene der Übersicht dargestellt und können durch einen Anwender – dieser kann auch mit dem Endnutzerentwickler der vorangegangenen Stufe identisch sein – genutzt werden. Die Anzahl der möglichen Variationen nimmt in dieser Übersicht von oben nach unten ab, auf der untersten Stufe steht wie erwähnt eine konkrete Anwendung.

Zum einen ist anzumerken, dass sich diese Erläuterungen nur auf Definitionen von Rollen beziehen, diese müssen nicht zwingend durch verschiedene Individuen ausgefüllt werden. Zum anderen sind unter Verwendung des Cobamos-Frameworks auch andere Entwicklungen als die in dieser Arbeit diskutierten möglich. Bei solchen Projekten würden sich in Abb. 1.2 entsprechend unterhalb der obersten Ebene die erstellten Artefakte und die jeweils zugehörigen Rollen verändern, die dargestellte Übersicht ist nicht allgemein gültig für alle denkbaren Verwendungen des Cobamos-Frameworks.

Im Zusammenhang mit dem erwähnten Begriff des Endnutzers ist eine kurze Klärung der Begrifflichkeiten erforderlich. Der Begriff des **Nutzers** ist in Abhandlungen über Themen der Informatik überbesetzt. Neben dem **menschlichen Nutzer** – beispielsweise einer Softwareanwendung – kann der Begriff Nutzer auch einen **Anwendungsteil** mit nicht näher bestimmten Eigenschaften bezeichnen, in dem eine an anderer Stelle bereitgestellte Funktionalität verwendet wird (der Anwendungsteil ist Nutzer der Funktionalität). Im weiteren Verlauf der vorliegenden Arbeit wird der Begriff des Nutzers häufig in dieser zweiten Bedeutung verwendet. Da aber immer wieder auch menschliche Nutzer Gegenstand der Betrachtungen sind, werden an Stellen, an denen die beabsichtige Bedeutung aus dem Zusammenhang nicht ersichtlich ist oder die Gefahr von Verwechselungen besteht, Umschreibungen der Form „menschliche Nutzer“ oder „nutzende Anwendungsteile“ gebraucht.

1.2 Aufbau der Arbeit

Die vorliegende Arbeit lässt sich im Wesentlichen in drei aufeinander aufbauende Teilbereiche – diese entsprechen den drei oberen Schichten in Abb. 1.2 – gliedern. Die Grafik in Abb. 1.3 veranschaulicht diesen Aufbau der Ausarbeitung. Jedes der Kapitel beginnt mit einer kurzen Zusammenfassung des Inhaltes, um eine schnellere Orientierung innerhalb der Ausarbeitung zu ermöglichen.

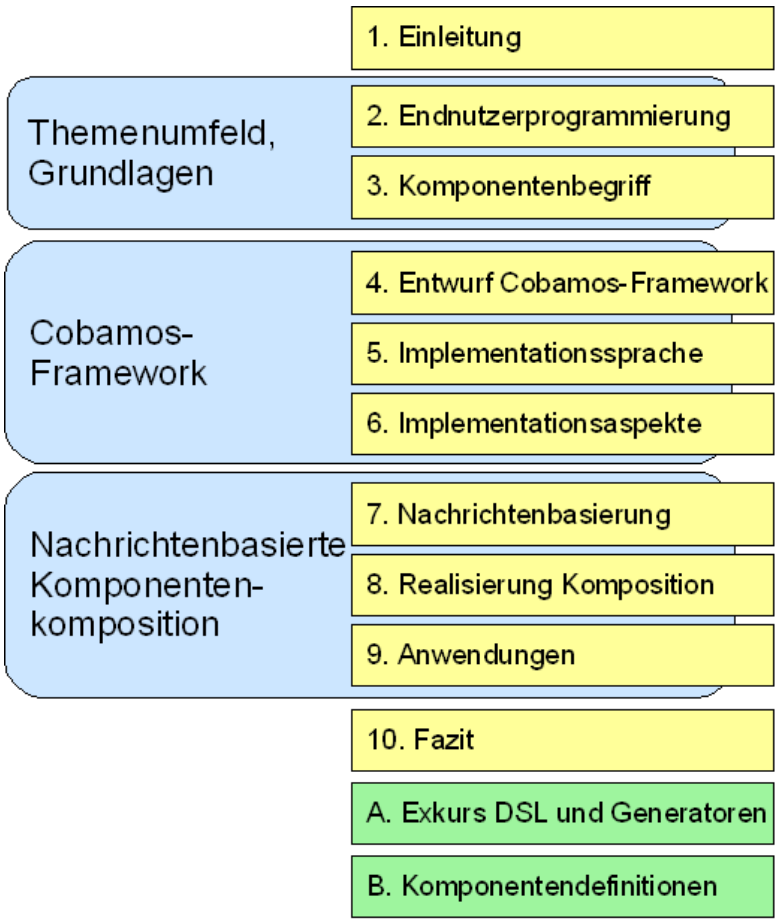


Abb. 1.3: Gliederung der vorliegenden Ausarbeitung

Nach dem Einleitungskapitel, welches der Motivation des Themas dient, werden im ersten Abschnitt der Arbeit zunächst das Themenumfeld und einige im weiteren Verlauf benötigte Grundlagen dargelegt. In Kapitel 2 wird der Themenkomplex der Endnutzerprogrammierung vorgestellt. Bei den Ansätzen, die in dieser Arbeit vorgestellt werden, finden in hohem Maße Komponenten Verwendung. Daher wird dieser Begriff im 3. Kapitel diskutiert. Die Betrachtungen bewegen sich dabei auf einer konzeptuellen Ebene, um eine von konkreten technologischen Umsetzungen unabhängige Definition zur Verfügung zu stellen.

Den zweiten Abschnitt der Ausarbeitung bilden die Kapitel rund um das Cobamos-Framework. In Kapitel 4 wird der eigentliche Entwurf beschrieben, eine Festlegung auf eine konkrete Implementation wird soweit als möglich vermieden. Die Betrachtung einiger ausgewählter Aspekte der Realisierung des Frameworks findet sich im Kapitel 6, zuvor wird jedoch die Auswahl der zur Umsetzung des Entwurfs verwendeten Programmiersprache C# in Kapitel 5 motiviert.

Nachdem damit eine Basis zur Realisierung von Endnutzerprogrammierumgebungen zur Verfügung steht, wird diese zur Realisierung einer Applikation zur nachrichtenbasierten Komposition von Komponenten verwendet. Die Grundlagen für dieses Vorgehen werden in Kapitel 7 dargestellt, dabei kann der Bereich der nachrichtenbasierten Kommunikation aufgrund des durch die Arbeit vorgegebenen Rahmens nicht in vollem Umfang dargestellt werden. Die Abhandlung orientiert sich daher an den im Weiteren benötigten Aspekten, gibt jedoch Ausblicke auf interessante Erweiterungsmöglichkeiten. Kapitel 8 widmet sich dem Entwurf und der Realisierung unter Verwendung der Ansätze aus den vorangegangenen Kapiteln. Den Abschluss des dritten Abschnittes bildet die Vorstellung exemplarischer Anwendungen, welche die Umsetzbarkeit der bis zu diesem Zeitpunkt beschriebenen Überlegungen zeigen. Bei diesen Anwendungen handelt es sich um die oben angesprochenen Anwendungsfamilien.

Der Kerntext der Ausarbeitung endet mit einer Zusammenfassung und einem Fazit in Kapitel 10.

Im Anhang zu dieser Arbeit ist zum einen ein Exkurs zu den Themen Anwendungsfamilien, domänenspezifische Sprachen und Softwaregeneratoren zu finden, da diese an einigen Stellen Berührungspunkte mit den in dieser Arbeit vorgenommenen Überlegungen haben. Zum anderen werden die bei den Recherchen zum Komponentenbegriff untersuchten Literaturzitate zur Verfügung gestellt.

2 Endnutzerprogrammierung

„I think there is a world market for maybe five computers.“
(Ich denke, dass es einen Weltmarkt für vielleicht fünf Computer gibt.)
Thomas J. Watson, Gründer der IBM, 1943, Echtheit umstritten

In der Zeit, aus welcher das Eingangszitat dieses Kapitels stammen soll, war Programmentwicklung durch Anwender, durch Nichtexperten undenkbar. Schon der einfache Umgang mit Rechenanlagen war Spezialisten vorbehalten. Heute ist der Umgang mit leistungsfähigen Rechnern in vielen Bereichen zur Selbstverständlichkeit geworden und das Thema Endnutzerprogrammierung ist Gegenstand der Forschungen in der Informatik.

Da auch die vorliegende Arbeit einen Beitrag in diesem Kontext liefern soll, gibt das vorliegende Kapitel zunächst eine Definition für den Begriff Endnutzerprogrammierung und diskutiert anschließend einige damit einhergehende Vor- und Nachteile. Vor diesem Hintergrund wird eine Möglichkeit der Klassifikation von Arbeiten im Bereich der Endnutzerprogrammierung aufgezeigt und die vorliegende Arbeit entsprechend eingeordnet. Den Abschluss des Kapitels bildet die exemplarische Vorstellung zweier Ansätze aus dem Bereich der Endnutzerprogrammierung. Während die formularbasierten Systeme, deren prominentester Vertreter die Tabellenkalkulationen sind, weitgehend bekannt und vielerorts im Einsatz sind, ist der zweite Ansatz, das Programmieren durch Beispiele, ein recht selten außerhalb des akademischen Umfeldes eingesetztes Konzept.

2.1 Einführung und Begriffsdefinition

In den letzten 20 bis 30 Jahren hat sich das technische Umfeld, in dem ein Großteil der Arbeitnehmer seine täglichen Aufgaben verrichtet, grundlegend verändert. Der Computer, früher in Form eines Großrechners ausschließlich die Domäne von Spezialisten, ist nun ein selbstverständlicher Bestandteil der täglichen Arbeit. Mit der zunehmenden Computerisierung sind auch die Kenntnisse und Fähigkeiten der Anwender im Umgang mit Computern respektive Software gestiegen.

Der Umgang mit Anwendungsprogrammen, die dem Nutzer erlauben in einem gewissen Rahmen neue, eigene Funktionalität zu definieren, ist in vielen Bereichen so selbstverständlich, dass viele dieser Tätigkeiten nicht bewusst als Programmieren wahrgenommen werden. In diesen Bereich gehören beispielsweise die Definition von Filtern in E-Mail-Anwendungen, die Realisierung von Anwendungen auf Basis von Tabellenkalkulationen, das Erstellen von interaktiven Webseiten mithilfe so genannter Wizards – Hilfsprogramme, die den Anwender durch eine Aufgabe führen – bis hin zur Entwicklung von Makros[2] in Anwendungen, die dies unterstützen (z. B. die Office Produkte der Firma Microsoft). Diese Tätigkeiten haben alle gemeinsam, dass durch Anwender, die nicht notwendigerweise in der Entwicklung von Software ausgebildet sind, neue Funktionalität geschaffen bzw. vorhandene angepasst wird. Die resultierenden Anwendungen werden in Unternehmen zum Teil zur Entscheidungsfindung in betrieblichen Abläufen oder allgemeiner zur Unterstützung von Managementaufgaben verschiedenster Art eingesetzt und haben damit Einfluss auf Geschäftsprozesse in den betreffenden Unternehmen. Dies macht deutlich, welche praktische Relevanz die Forschungen und Entwicklungen in diesem Bereich haben.

[2] Makros sind in diesem Zusammenhang kleinere Programme, welche der Automatisierung von Aufgaben dienen. Als Entwicklungssprache dient innerhalb der Microsoft Office Familie ein BASIC Dialekt, Visual Basic for Applications (VBA).

Aufgrund der angesprochenen Entwicklungen sind die Möglichkeiten der Unterstützung von Anwendern bei solchen Tätigkeiten durch geeignete organisatorische Strukturen, individuelles Umfeld und entsprechende Applikationen ein seit einigen Jahren stetig wachsendes Forschungsgebiet. Das Spektrum reicht hier von Überlegungen zur Qualität [Ala85], über Programmierparadigmen [BAD+01] und Kostenabschätzungen [Sut05] bis zu neueren Entwicklungen im Bereich mobile Systeme [KR06].

Für dieses Forschungsgebiet finden sich in der aktuellen Literatur die Begriffe **End User Computing (EUC)**, **End User Programming (EUP)** und **End User Development (EUD)**, häufig auch synonym verwendet. Wie in vielen Bereichen der Informatik existieren auch an dieser Stelle unterschiedliche Definitionen. Daher soll nun für den weiteren Verlauf dieser Arbeit eine Definition des Begriffes Endnutzerprogrammierung gegeben werden.

In der Forschungsliteratur lassen sich zwei grundlegend verschiedene Definitionen unterscheiden (siehe [Dow04], [PM02] für entsprechende Literaturverweise). Zum einen wird unter dem Begriff End User Computing jedweder Umgang mit Informationstechnologie durch Endanwender gesehen. Diese Definition ist sehr weit gefasst und geht über den Rahmen hinaus, auf den diese Arbeit abzielt. Zum anderen wird der Umgang von Nichtexperten mit Informationssystemen mit dem Ziel der Entwicklung von Softwareanwendungen zur Durchführung von Aufgaben innerhalb einer Organisationseinheit oder zur Unterstützung von Entscheidungsprozessen unter dem Begriff End User Computing subsumiert. Es erscheint nahe liegend, den Begriff End User **Computing** ausschließlich für die weiter gefasste Definition und End User **Development** bzw. **Programming** für die eingegrenztere zweite Definition zu verwenden. Damit ist das End User Development/Programming eine Teilmenge des End User Computing.

Diese eingrenzende Sichtweise entspricht eher den Zielen dieser Arbeit und motiviert folgende Definition:

> *Endnutzerprogrammierung bezeichnet im Rahmen dieser Arbeit die Entwicklung bzw. Anpassung von Softwaresystemen zur Lösung von Problemen aus dem eigenen Anwendungsgebiet durch Personen, die nicht im Bereich der Softwareentwicklung ausgebildet oder professionell tätig sind.*

Diese Definition verzichtet auf die Einschränkung hinsichtlich der verfolgten Ziele einer solchen Entwicklung, betont aber die Anwendung von Techniken der Informatik bzw. genauer der Softwareentwicklung durch Nichtexperten zur Lösung von Problemen aus ihrem eigenen Anwendungsbereich.

Die eventuelle nahe liegende Definition, Endnutzerprogrammierung als die Entwicklung durch die späteren Anwender zu sehen, würde Fälle ausschließen, in denen Problemlöser und Anwender nicht identisch sind. Dies ist beispielsweise der Fall, wenn ein Dozent eine individualisierte Lernanwendung erstellt, diese aber nicht von ihm sondern ausschließlich durch Schülerinnen und Schüler genutzt würde.

Die folgenden Abschnitte diskutieren einige der Vor- und Nachteile der Endnutzerprogrammierung, um vor diesem Hintergrund eine bessere Einschätzung der Arbeit und der später diskutierten Ansätze zu ermöglichen.

2.2 Vorteile

Einer der am häufigsten zitierten Vorteile der Softwareentwicklung durch Endanwender ist die Reduktion der so genannten **expertise tension** [Pic05] [HP02], des Spannungsverhältnisses zwischen

der Expertise in einer Anwendungsdomäne und der in Entwicklung von Softwareanwendungen. Anwender werden in der Regel als Experten ihrer jeweiligen Domäne angesehen, verfügen aber nur selten über weit gehende Kenntnisse in der Anwendungsentwicklung. Anwendungsentwickler hingegen verfügen in den meisten Fällen über zu wenige Kenntnisse einer konkreten Anwendungsdomäne um eine fachlich anspruchsvolle Anwendung konzipieren zu können.

Bei der Entwicklung einer Applikation stellt der notwendige Wissenstransfer von den Anwendern hin zu den Entwicklern, welcher während der Analysephase durchgeführt werden muss, die problematischste Phase im Projektverlauf dar. Die Evaluierung und Dokumentation von Wissen stellt Anforderungen an Entwickler, die nicht ihren Kernkompetenzen entsprechen. Zudem haben Fehler in dieser Phase schwer wiegende Auswirkungen auf das gesamte Projekt, bis hin zur Entwicklung einer Applikation, die nicht den Anforderungen bzw. Vorstellungen der späteren Anwender entspricht. Dieses Problem wird bei von Endnutzern entworfenen Anwendungen vermieden.

Doch selbst wenn davon ausgegangen wird, dass bei der Ermittlung des durch die Anwender ausgedrückten Wissens keine Fehler durch die Softwareentwickler bzw. Analytiker gemacht werden, besteht die Gefahr, nicht das gesamte auf der Anwenderseite verfügbare Domänenwissen erfassen zu können. Unter dem Begriff **tacit knowledge** (stilles Wissen) wird in der Literatur zur Endnutzerprogrammierung das Phänomen diskutiert, dass Anwender über mehr Wissen verfügen, als sie ausdrücken können (siehe z. B. [Wag02]). Der Schritt der Wissensvermittlung von den Anwendern hin zu den Entwicklern scheitert in diesem Fall nicht am mangelnden Domänenwissen oder an Analysefehlern oder Missverständnissen auf Seiten der Entwickler, sondern daran, dass die Nutzer Wissen, welches sie täglich in ihren Arbeitsprozessen anwenden, nicht ausdrücken können bzw. sich dieses impliziten Wissens gar nicht bewusst sind. Für die Anwendungsentwickler ist es nur schwer möglich, dieses Wissen aufzudecken und explizit zu dokumentieren.

Ein Beispiel für dieses nicht explizit vorhandene bzw. ausdrückbare Wissen bei Domänenexperten findet sich bei [Wat86] im Zusammenhang mit der Konstruktion von Expertensystemen. In diesem Beispiel wird eine Versicherungssachbearbeiterin im Rahmen der Analysephase für ein Expertensystem um eine Schätzung der zukünftigen Gesamtkosten eines Versicherungsfalls gebeten und gibt diese nach kurzem Überlegen ab. Auf die Frage nach den zur Bestimmung des Betrages verwendeten Regeln und Berechnungen kann keine Antwort gegeben werden, das Wissen ist nicht explizit formulierbar und die Schätzung wird mit der vorhandenen Erfahrung im Anwendungsgebiet begründet. Aufgrund der Nachfragen des Expertensystementwicklers werden die Annahmen schrittweise explizit dargestellt und erläutert. Am Ende der daraus resultierenden – recht umfangreichen – Berechnungen steht ein nur wenig (< 1%) von der Schätzung abweichender Betrag.

Von der Endnutzerprogrammierung erhofft man sich in diesem Zusammenhang, dass ein Endnutzer mit der Möglichkeit ein Softwaresystem selbst zu entwerfen dieses stille Wissen beim Entwurf der Anwendung nutzt und dadurch explizit verfügbar macht. Selbst wenn diese Applikation nur als Prototyp einer Entwicklung durch Softwareentwickler verwendet wird, ermöglicht sie eine genauere Untersuchung des Wissens durch die Anwendungsentwickler, als dies in Interviews erreicht werden kann. Darüber hinaus bietet der Prototyp die Möglichkeit der – anderenfalls aufgrund des mangelnden Domänenwissens bei den Entwicklern schwierigen – Validation und Verifikation[3] des von den Entwicklern ermittelten und umgesetzten Domänenwissens. Berechnungsvorschriften müssen beispielsweise zum selben Ergebnis führen, Schritte in Arbeitsabläufen dieselben Vor- bzw. Nachbedingungen haben.

[3] Die Validation kann aufgrund des Vergleiches der durch die Analyse beschriebenen und im Prototypen umgesetzten Funktionalitäten unterstützt werden. Im Prototypen umgesetzte Funktionalitäten (z. B. Berechnungen) können durch Vergleich der Wirkungen der Verifikation dienen.

Endnutzer, die Anwendungen selbst entwickeln oder zumindest in einem vorgegebenen Rahmen anpassen können, sind in der Lage flexibler auf Veränderungen in ihrer Arbeitswelt zu reagieren, die Änderungen an den eingesetzten Informationssystemen bedingen.

Da die Wartungsphase einer Softwareanwendung – die Phase, in der solche Anpassungen vorgenommen werden – einen beträchtlichen Kostenfaktor darstellt, kann ein Einbinden der Endnutzer in diesen Prozess zu einer Reduktion der Kosten führen. Durch die Endnutzer ausführbare Anpassungen müssten nicht mehr an externe Spezialisten vergeben werden. Die Elimination der anderenfalls notwendigen Kommunikationsschritte ermöglicht darüber hinaus eine Verringerung der Gesamtdauer der Maßnahme. Dem stehen notwendige Aufwendungen für die entsprechende Ausbildung der Endnutzer, benötigte Ressourcen und Ähnliches entgegen. Eine genaue Abwägung kann hier nur im Rahmen detaillierter betriebswirtschaftlicher Betrachtungen erfolgen; diese würde jedoch den Rahmen der vorliegenden Arbeit sprengen.

Unabhängig von diesen direkten Kostenbetrachtungen legen Untersuchungen nahe, dass es insgesamt einen Produktivitätszuwachs durch den Einsatz von Endnutzerprogrammierung in einer Organisation geben kann [BH98] [WJ04].

Neben den eben angesprochenen ökonomischen Vorteilen erwachsen weitere Vorteile aus der stärkeren Einbindung der Nutzer in die Entwicklung einer Anwendung. Dies führt in der Folge zu einer höheren Akzeptanz der Applikation und zu einer besseren Performanz des Nutzers mit der Anwendung. Solche Effekte sind auch durch empirische Studien (vgl. [McG04]) belegt worden. Allerdings muss einschränkend bemerkt werden, dass in der Studie mehrere Endanwender Anwendungen auf Basis einer Tabellenkalkulation entwarfen und die Effekte nur für die jeweils eigene Applikation nachweisbar waren.

In der oben angeführten Studie wurde von den Teilnehmern einer eigenentwickelten Applikation in der Regel eine höhere Qualität bescheinigt als Anwendungen, die von anderen Nutzern mit gleicher Qualifikation erstellt wurden. Empirische Untersuchungen legen nahe, dass die Bewertung einer eigenentwickelten Applikation hinsichtlich der Anwendungsqualität durch Endnutzer weniger objektiv erfolgt als die Bewertung einer „fremden“ Anwendung[4]. In [McG02] wird eine deutliche Abweichung zwischen der Selbstbewertung von endnutzerentwickelten Anwendungen durch den entwickelnden Endnutzer und der Bewertung unabhängiger Softwareexperten nachgewiesen. Dieser Umstand deutet auf ein Problem der Endnutzerprogrammierung hin. Gerade Endnutzer mit einem niedrigen Qualifikationsgrad rechnen ihrer Anwendung eine zu hohe Qualität zu. Sie setzen diese aber entsprechend ihrer Einschätzung in der täglichen Arbeit ein. Der folgende Abschnitt beschäftigt sich genauer mit den daraus resultierenden Problemen und anderen Nachteilen.

2.3 Gefahren und Nachteile

Wie eingangs bereits erwähnt, finden die im Rahmen der Endnutzerprogrammierung entstehenden Applikationen Einsatz in diversen betrieblichen Abläufen. Aufgrund der mit diesen Anwendungen ermittelten Ergebnisse werden unternehmerische Entscheidungen gefällt, mit den Anwendungen werden Daten manipuliert, deren Konsistenz sichergestellt sein muss, und Ähnliches. Dies alles geschieht, obwohl die Anwendung von einem Laien in der Softwareentwicklung, von einem „Dilettanten“ [Har04] entwickelt wurde.

[4] Dies erscheint analog zu der teilweise zu beobachtenden Ablehnung der Wiederverwendung von fremdentwickelten Bibliotheken und Komponenten durch Softwareentwickler. Dieser Effekt wird als „not invented here“-Syndrom bezeichnet, siehe [Zwi05].

In den letzten Jahren sind verstärkt Sicherheitslücken in Softwareanwendungen und die daraus resultierenden Folgen in den Blickpunkt des öffentlichen Interesses geraten. Dabei handelt es sich in der Regel um Sicherheitslücken in Applikationen, die von geschulten Anwendungsentwicklern realisiert wurden. In der Endnutzerprogrammierung werden Applikationen von Anwendern realisiert, die nicht über eine fundierte Ausbildung im Bereich Softwareentwicklung verfügen. Techniken zum systematischen Testen und andere Ansätze der Qualitätssicherung, die im Software Engineering bereits etabliert sind, gehören nur in Ausnahmefällen zum Wissen der Endnutzer. Diese Überlegungen lassen vermuten, dass solche Anwendungen fehlerbehaftet sein können und damit ein Risiko beispielsweise im oben angesprochenen betrieblichen Einsatz darstellen. In der Tat existieren einige Berichte – insbesondere für Anwendungen auf Basis von Tabellenkalkulationen – über beträchtliche Schäden, welche Unternehmen und Organisationen durch fehlerhafte Endnutzeranwendungen bzw. den falschen Umgang mit solchen Anwendungen entstanden sind. Diese reichen von fälschlicherweise vermuteten Manipulationen an Notenspiegeln durch externe Hacker am MIT [Neu00] bis zu einem auf 24 Millionen US-$ geschätzten Schaden für einen Stromerzeuger auf dem amerikanischen Kontinent [Tra03] [Neu03]. Eine Zusammenstellung weiterer Vorfälle findet sich unter [Eur06] und [EUS05]. Es ist interessant, dass in den betroffenen Unternehmen offensichtlich keine internen Zertifizierungsmechanismen existierten, die solche Vorfälle hätten verhindern können.

Es gibt verschiedene nahe liegende Ansätze für eine Verbesserung dieser Situation. Zum einen kann die Werkzeugunterstützung für Endanwender bei der Entwicklung von Anwendungen, insbesondere im Bereich der Sicherung der Softwarequalität und der Fehlervermeidung, verbessert werden. Zum anderen muss man sich die Frage stellen, ob solche Anwendungen in kritischen Bereichen ohne weiteres eingesetzt werden dürfen. Denkbar wären beispielsweise veränderte Prozesse zur Erstellung von endnutzerentwickelten Applikationen, die eine Abnahme – im Sinne der oben angesprochenen Zertifizierung – durch geschulte Experten voraussetzen. Eine andere Möglichkeit stellt eine Erhöhung der Qualifikation der Endnutzer in Teilbereichen der Softwareentwicklung dar. In [BCR04] [RB05] wird beispielsweise ein als „End User Software Engineering“ bezeichneter Ansatz vorgestellt, der einige dieser Überlegungen kombiniert. Es wird eine Kombination aus interaktivem Testen, Werkzeugunterstützung bei der Fehlerlokalisierung, der Möglichkeit Zusicherungen zu nutzen und der Wissensvermittlung durch die Anwendung beschrieben.

Es gibt eine Reihe von Anwendungsbereichen – von denen einige bereits angeklungen sind, andere werden später in dieser Arbeit als mögliche Einsatzfelder für die erarbeiteten Konzepte diskutiert – in denen Endnutzerprogrammierung nutzbringend eingesetzt werden kann und Fehler in der Anwendung nur ein beschränktes Risiko darstellen. In anderen Bereichen sind höhere Qualitätsanforderungen an eingesetzte Softwareapplikationen zu stellen. In diesen müssen die aus der Endnutzerprogrammierung erwachsenden Risiken bewusst wahrgenommen und eingeschätzt werden. Oder wie Warren Harrison von der Portland State University in [Har04] formulierte: *„Can it be true that software manipulating my credit history could have been written by an accountant with no concept of software testing or development processes?“*[5] Die Antwort auf diese Frage sollte – und dies kann für jede in einem kritischen Bereich eingesetzte Anwendung verallgemeinert werden – „nein“ sein.

Neben dem Risiko von Fehlern in den Anwendungen entstehen aus den mangelnden Kenntnissen in der Entwicklung von Softwareanwendungen weitere Probleme. Häufig fehlt die Dokumentation zu den Anwendungen [Pic05], das entsprechende Wissen ist nur im Kopf des Endnutzers vorhanden. In der Regel werden unternehmensinterne Entwicklungsstandards nicht befolgt, Versionsverwaltung und

[5] „Kann es wahr sein, dass Software, welche meine Kredithistorie manipuliert, von einem Buchhalter geschrieben worden sein könnte, der keinen Begriff von systematischem Testen oder Entwicklungsprozessen hat?“

Änderungsverfolgung existieren nicht, Datenformate werden nur unzureichend dokumentiert. Diese Liste lässt sich weiter fortsetzen. Alle diese Punkte tragen dazu bei, dass die aus der Endnutzerprogrammierung resultierende Anwendung nur schlecht wartbar ist und verringern ihre Qualität. Die Probleme, die im Software Engineering für die professionelle Entwicklung von Softwareanwendungen bekannt sind und zu deren Behebung eine ganze Reihe von Techniken und Theorien entwickelt wurden, existieren vielfach in kleinerer Form in der Endnutzerprogrammierung.

Die eben angesprochene fehlende Dokumentation stellt darüber hinaus ein Problem dar, wenn andere Anwender als der entwickelnde Endnutzer eine Anwendung einsetzen sollen. Je nach Komplexität der Applikation entsteht an dieser Stelle Schulungsbedarf, der aufgrund der fehlenden Dokumentation nur durch den entwickelnden Endanwender zu decken ist.

Weitere potenzielle Probleme ergeben sich aus der mangelnden Kenntnis einschlägiger rechtlicher Vorschriften. So mag es beispielsweise für einen Abteilungsleiter mit entsprechenden Kenntnissen in den heute üblichen Büroanwendungen nahe liegen, die Fehlzeiten seiner Angestellten durch eine Anwendung auf Basis einer Tabellenkalkulation zu verwalten. Da hierbei jedoch personenbezogene Daten verarbeitet werden, müssen entsprechende Vorgaben aus Bundes- beziehungsweise Landesdatenschutzgesetz beachtet werden. Dieses Wissen fehlt jedoch häufig (siehe z. B. [Bay04] [Auf05]).

Da Endnutzerentwicklungen in der Regel nicht in Form eines Projektes durchgeführt werden und eine Person die gesamte Erstellung der Applikation von der Problemanalyse bis zur Realisierung durchführt, bilden sich keine Arbeitspakete, die sich parallel entwickeln ließen. Daher ist zu erwarten, dass Endnutzerprogrammierung schlecht skaliert. Die Verzahnung von Teilprojekten und die Gestaltung notwendiger Schnittstellen zur späteren Integration einzelner Anwendungsteile setzt ein Wissen aus dem Bereich der Softwareentwicklung voraus, welches definitionsgemäß bei der Endnutzerprogrammierung nicht vorhanden ist. Es bleibt daher festzuhalten, dass die umsetzbaren Anwendungen einen Umfang haben, der von einer Person in einer überschaubaren Zeitspanne bewältigt werden kann.

Nach der Betrachtung der Vor- und Nachteile, die aus der Endnutzerprogrammierung erwachsen, dient der nächste Abschnitt der Einordnung der vorliegenden Arbeit innerhalb des Forschungsgebietes der Endnutzerprogrammierung.

2.4 Einordnung dieser Arbeit

2.4.1 Einordnung nach Teilgebieten der Forschungsliteratur

Es existieren mehrere Arbeiten [Dow04] [PM02] [Har00] [BB93], die mit zum Teil unterschiedlichen Schwerpunkten eine Einteilung der zum jeweiligen Zeitpunkt vorhandenen Literatur zum Thema Endnutzerprogrammierung und der angrenzenden Forschungsgebiete vornehmen. Die gewonnenen Einteilungen unterscheiden sich leicht in ihrer Ausrichtung und ihrem Detaillierungsgrad. Für die Einordnung dieser Arbeit wird das in [Dow04] vorgestellte **EUC Research Framework** (siehe Abb. 2.1) verwendet, welches im Folgenden kurz vorgestellt werden soll.

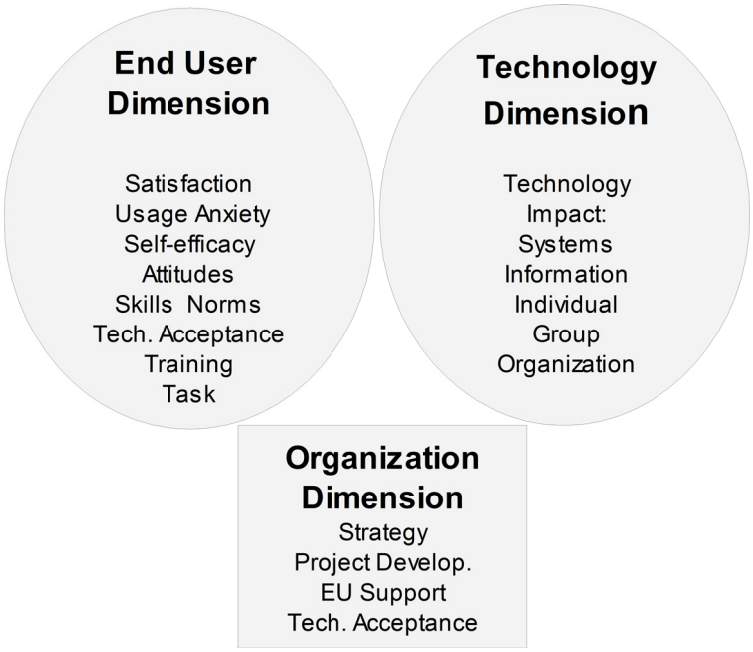


Abb. 2.1: EUC Research Framework nach [Dow04]

Diese Taxonomie teilt die Forschungsliteratur im Bereich des End User Computing naheliegenderweise nach dem Schwerpunkt der jeweiligen Studie ein. Dabei wird unterschieden, ob sich die Arbeit auf den Endnutzer selbst, auf die technologische Ebene oder auf das organisatorische Umfeld konzentriert. Die Teilbereiche werden als Endnutzerdimension (End User Dimension), Technologiedimension (Technology Dimension) und Organisationsdimension (Organisation Dimension) bezeichnet.

Diese drei Teilbereiche entsprechen im Wesentlichen den auch in [Har00] identifizierten drei „Schools of Thought“ (Behavioural, Application, Organisation). Die untersuchte Literatur dieser Studie war auf Arbeiten beschränkt, welche Einflüsse auf den Erfolg des End User Computing untersuchten, dennoch ergab sich eine vergleichbare Einteilung.

In die Endnutzerdimension fallen in der vorgestellten Systematik alle Arbeiten, welche sich auf das Individuum, also den Endnutzer als Person, beziehen. Darunter sind Studien zur Nutzerzufriedenheit, zum Verhalten von Endnutzern, zur Akzeptanz von Technologien und Ähnliches subsummiert. Studien, die sich hauptsächlich mit der Anwendung von Technologien für die Endnutzerprogrammierung auseinandersetzen, werden der Technologiedimension zugeordnet. Voraussetzung ist jedoch der Bezug zur Endnutzerprogrammierung, beispielsweise die Untersuchung des Einflusses auf den Endnutzer. Die Grafik zeigt die in der Studie unterschiedenen Ebenen, auf denen eine Technologie Einfluss haben kann (System- und Informationsqualität, Individuum, Gruppe, Organisation). Reine Technologiestudien wurden laut [Dow04] nicht berücksichtigt. Hier ist anzumerken, dass die Trennung in die einzelnen Bereiche teilweise unscharf ist, eine Studie über die Akzeptanz und Eignung einer neuen Technologie berührt sowohl die Endnutzer- als auch die Technologiedimension. Die verbleibende Dimension der Arbeiten mit Schwerpunkt auf der Untersuchung des organisatorischen Umfeldes umfasst unter anderem Arbeiten, die sich mit dem Management von Endnutzerprogrammierung, dem Einfluss der Endnutzerprogrammmierung auf die individuelle Effizienz oder der Gestaltung geeigneter Entwicklungsprozesse auseinandersetzen.

In diesem Framework ist die vorliegende Arbeit in der Technologiedimension zu positionieren, wobei eine Betrachtung der Auswirkungen auf den Endnutzer auf konzeptioneller Basis erfolgt. Nach

Abschluss dieser Arbeit könnte zur weiteren Unterstützung der Ergebnisse eine empirische Untersuchung erfolgen.

2.4.2 Einordnung nach Zielgruppe

Ausgehend von der Betrachtung der Zielgruppe kann eine weiter verfeinerte Einordnung einer Arbeit im Kontext der Endnutzerprogrammierung vorgenommen werden. Da die oben angesprochene Einteilung in drei Dimensionen recht grob ist, ist eine weitere Differenzierung der Technologiedimension sinnvoll.

In [CK89] [Gov03] wird eine Einteilung von Endanwendern entlang von drei Dimensionen diskutiert. Das Ziel der ursprünglichen Studie war eine Einteilung aller Personen oder Gruppen, die mit einem „computer-based information system“ [CK89] interagieren. Dafür wurden Endnutzer in den Dimensionen Operation, Entwicklung und Steuerung klassifiziert (siehe Abb. 2.2). Dabei ist unter Operation der Umgang mit Hard- oder Software zu verstehen, maßgeblich für die Bewertung in der Dimension Kontrolle ist Entscheidungs- und Führungskompetenz.

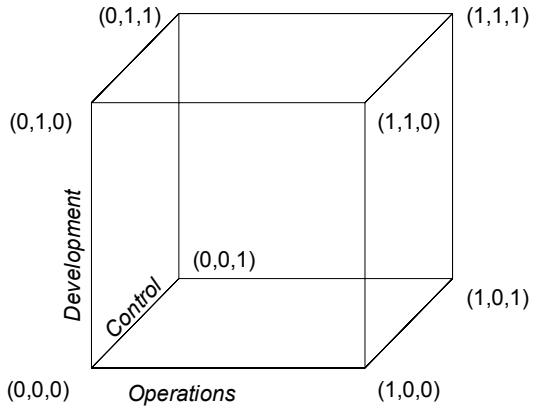


Abb. 2.2: Der „User Cube“ nach [CK89]

Da die vorliegende Arbeit auf die Endnutzerprogrammierung abzielt, ist damit eine implizite Eingrenzung der Endnutzer verbunden. Es werden ausreichende Fähigkeiten im Umgang mit einem Softwaresystem vorausgesetzt, ohne dass diese auf dem Level eines Administrators (hohes Wissen um den Umgang mit Hard- und Software entspräche der Ecke (1,0,0) im User Cube) liegen müssen. Dadurch wird bildlich gesprochen eine Ebene (eigentlich ein Quader, da der Bereich der Fähigkeiten nicht auf einen Punkt festgelegt werden kann, sondern ein Intervall auf der Operations-Achse umfasst) aus dem Würfel selektiert. Die Dimension der Führungskompetenz des Endnutzers spielt im Rahmen dieser Untersuchungen keine Rolle, an ihre Stelle tritt die Kompetenz hinsichtlich des Anwendungsbereiches, in welchem die Endnutzeranwendung angesiedelt ist.

Es ist sicherlich richtig, Endnutzern ein höheres Wissen um ihre Anwendungsdomäne zuzurechnen als einem beliebigen Softwareentwickler. Die in der Literatur teilweise zu findende Vereinfachung, dass alle Endnutzer Domänenexperten seien, ist bei genauerer Betrachtung jedoch zu grob. Vielmehr ist eine Differenzierung der Nutzer nicht nur hinsichtlich des Wissens um die Entwicklung von Softwareanwendungen möglich, auch bezüglich des Wissens im jeweiligen Anwendungsgebiet bestehen Unterschiede. Diese müssen sich in einer Anwendung zur Unterstützung der Endnutzerprogrammierung niederschlagen. Wenn das Wissen um den Anwendungsbereich nicht in

ausreichender Form vorhanden ist, kann eine Applikation beispielsweise durch unterstützende Dialoge – so genannte Wizards –, Vorschläge für Voreinstellungen, interaktive Hilfen und Ähnliches Unterstützung bieten. Ein Endnutzer mit einem hohen Wissensgrad im Anwendungsbereich wird eine Assistenz in dieser Form eher als störend empfinden[6].

Eine Anwendung, die einem Endanwender Unterstützung bei der Realisierung von Applikationen bieten soll, muss also den Kenntnisstand sowohl hinsichtlich der Softwareentwicklung als auch im Bezug auf die Anwendungsdomäne berücksichtigen. Damit ergibt sich eine Differenzierung der Anwender entlang der Dimensionen Entwurfswissen und Domänenwissen (siehe Abb. 2.3).

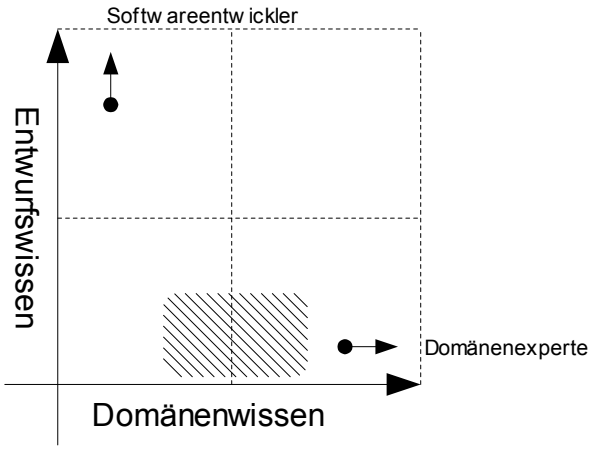


Abb. 2.3: Einteilung der Endnutzer entlang der Dimensionen Entwurfswissen und Domänenwissen

Softwareentwickler haben im Allgemeinen ein hohes Entwurfs- aber ein geringes Domänenwissen (oberer linker Quadrant), Domänenexperten haben ein hohes Wissen in ihrem Anwendungsbereich (unterer rechter Quadrant), in der Regel aber nur geringe oder gar keine Kenntnisse hinsichtlich der Entwicklung von Software.

Diese Einordnung ermöglicht – wie eingangs angesprochen – eine Klassifikation von Technologien oder konkreten Applikationen durch Betrachtung der adressierten Anwendergruppe. Domänenspezifische Sprachen (siehe Appendix A) dienen häufig dazu, Domänenwissen zu abstrahieren und für Nichtexperten verfügbar zu machen. Viele dieser Sprachen sind dennoch von der Anwendung her so komplex, dass sie für den Einsatz durch professionelle Entwickler und nicht durch Endanwender gedacht sind. Eine solche Sprache wäre also in dem oberen linken Quadranten anzusiedeln. Die Entwicklung einer Anwendung für den Controlling-Bereich auf Basis einer Tabellenkalkulation erfordert kein ausgeprägtes Entwurfswissen aber gute Kenntnisse des Anwendungsbereiches. Diese Tätigkeit würde also in den Bereich des unteren rechten Quadranten fallen.

Die in der vorliegenden Arbeit diskutierten Ansätze sollen in weitestem Sinne dazu dienen, Endanwendern die Realisierung von Softwareapplikationen ohne Hilfe eines Experten zu ermöglichen. Bezogen auf die oben diskutierte Klassifikation bedeutet dieses, sie sollen das benötigte Entwurfswissen senken. In der Zielgruppe wird ein gewisses Maß an Wissen über die

[6] Dieser unterschiedliche Bedarf an Unterstützung motiviert auch einige der Überlegungen hinsichtlich der Anpassbarkeit von Applikationen im weiteren Verlauf der Arbeit. Idealerweise ist eine Anpassung der Anwendung auf unterschiedliche Nutzergruppen möglich oder ein Anwender kann den für ihn notwendigen Unterstützungslevel wählen.

Anwendungsdomäne vorausgesetzt, es wird aber auch die Wiederverwendung von Domänenwissen unterstützt. Insgesamt erscheint eine Einordnung in den durch die Schraffur markierten Bereich sinnvoll.

2.5 Beispiele für Ansätze und Applikationen

Im Folgenden sollen beispielhaft zwei Programmierparadigmen der Endnutzerprogrammierung vorgestellt werden. Diese Aufzählung erhebt keinen Anspruch auf Vollständigkeit, vielmehr soll ein Eindruck von den Entwicklungen im Bereich der Endnutzerprogrammierung vermittelt werden. Studien, die sich auf die Untersuchung des Endnutzers oder des organisatorischen Umfeldes konzentrieren, werden dabei – da sie zu weit vom Kernthema dieser Arbeit entfernt sind – nicht vorgestellt. Einen guten Einstieg in das gesamte Forschungsgebiet Endnutzerprogrammierung inklusive der in dieser Arbeit nicht betrachteten Aspekte auf Basis der verfügbaren Literatur der letzten Jahre bieten [Har00] [PM02] und [Dow04].

2.5.1 Formularorientierte Programmiersysteme

Tabellenkalkulationen oder allgemeiner formularorientierte Programmiersysteme sind der am weitesten verbreitete Ansatz zur Realisierung von Applikationen durch Endnutzer (siehe auch [BCR04]). Viele andere Ansätze aus dem akademischen Bereich sind durchaus viel versprechend und innovativ, besitzen aber keine oder nur wenig Verbreitung über den Rahmen der jeweiligen Studien hinaus.

Dies führt dazu, dass die Entwicklung von Applikationen auf Basis von Tabellenkalkulationen der wohl am umfassendsten untersuchte Bereich innerhalb der Endnutzerprogrammierung ist. Insbesondere im Hinblick auf empirische Studien ist dies verständlich, hierbei ist eine gewisse Grundgesamtheit von Anwendern vonnöten um Aussagen treffen zu können. Aber auch Technologiestudien implementieren neue Techniken häufig prototypisch durch Erweiterung von Tabellenkalkulationen. Dies hat den Vorteil, dass die Endnutzer zumindest zum Teil mit einer für sie gewohnten Anwendung interagieren.

Eines der bekanntesten Produkte in diesem Bereich ist das von der Firma Microsoft vertriebene Produkt Excel. In der unten stehenden Abbildung (Abb. 2.4) ist eine Kreditberechnung auf Basis dieser Tabellenkalkulation zu sehen, diese gehört zum Lieferumfang der Anwendung. Nach Eingabe der Werte in der Tabelle „Werte eingeben“ werden alle übrigen angezeigten Werte berechnet. Diese Anwendung verdeutlicht die Art der Anwendungen, die mittels Tabellenkalkulationen umgesetzt werden, ist aber vergleichsweise wenig komplex und umfangreich.

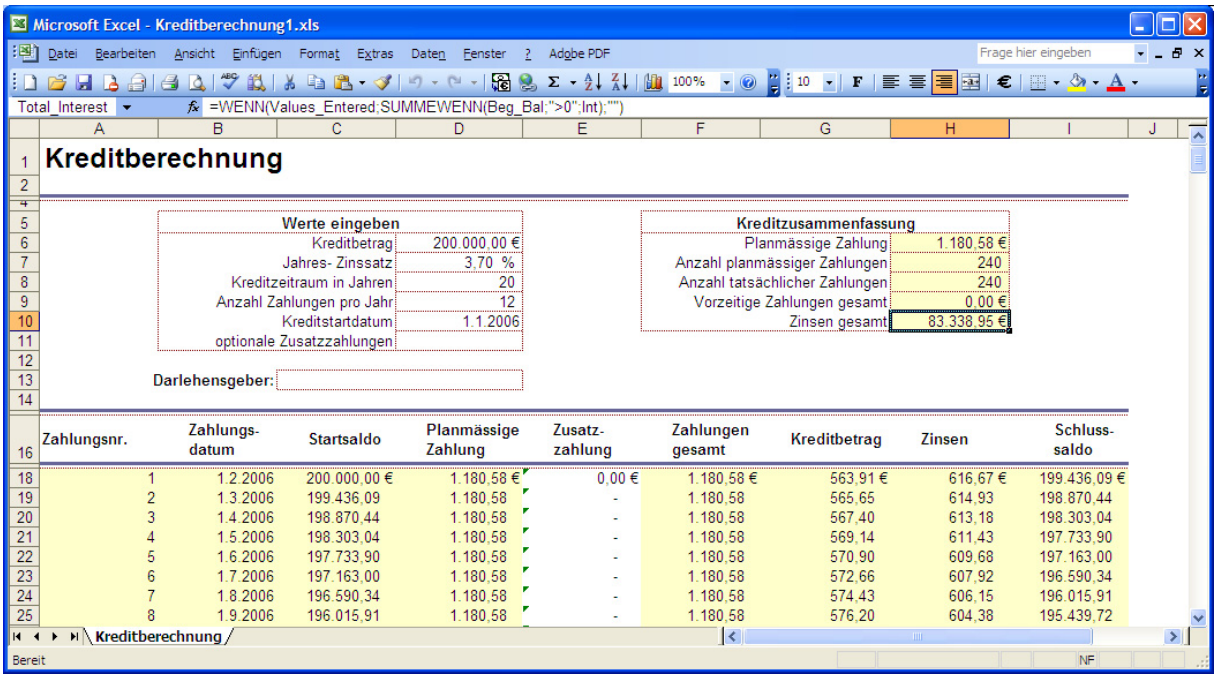


Abb. 2.4: Kreditberechnung mit Microsoft Excel

Solche in Tabellenkalkulationen entwickelten Applikationen unterliegen gewissen Einschränkungen, die aus der Anwendung des formularorientierten Paradigmas resultieren (vgl. [Sch98]). Anwendungen werden durch Ausfüllen einzelner Zellen in den Tabellenblättern erzeugt, einer Zelle können entweder Daten oder Formeln zugeordnet werden. Formeln werden dabei nicht direkt angezeigt, die Zelle enthält das Ergebnis der Berechnung, die in der Regel auf Basis anderer Zellen erfolgt. Die Zuweisung von Darstellungseigenschaften an die Zellen ist nützlich für die Gestaltung der Interaktion mit dem Anwender, hat aber keine funktionalen Auswirkungen. Da durch den Anwender bei Berechnungen nur vorgegeben wird, wie ein Wert durch eine Formel aus anderen Werten errechnet wird, also nur was getan werden soll, ist die Programmierung auf Basis von Formularen deklarativ. Es werden keine Algorithmen beschrieben, alle Berechnungen erfolgen zustandslos. Eine sequenzielle Verarbeitung wird nicht unterstützt, und es gibt keine Konzepte zur Realisierung von Kontrollstrukturen[7] im Sinne der prozeduralen Programmierung. Dabei ist anzumerken, dass die Effekte von Entscheidungen jedoch in der Regel auch durch so genannte „bedingte Berechnungen“ erreicht werden können. Veränderungen von Werten einzelner Zellen führen zur sofortigen Aktualisierung der über Formeln abhängigen Zellen. Größere Softwaresysteme können in Ermangelung von adäquaten Konzepten zur Datenabstraktion und Kapselung nicht umgesetzt werden.

Trotz dieser Einschränkungen sind Anwendungen auf dieser Basis sehr verbreitet. Im Abschnitt 2.3, Gefahren und Nachteile, wurde bereits angesprochen, welche Risiken aus Anwendungen dieser Art in Unternehmen entstehen können, und dass Fehler in solchen Anwendungen durchaus Auswirkungen über das lokale Umfeld des entwickelnden Endnutzers hinaus haben. Diese Tatsache motiviert viele Untersuchungen im Bereich der Unterstützung zur Fehlersuche und Fehlervermeidung in Tabellenkalkulationen. Die Arbeiten von Burnett, Cook und Rothermel [BCR04] wurden bereits angesprochen, eine andere Arbeit [MMM+02] konzentriert sich auf die Unterstützung der Suche nach Referenzierungsfehlern.

[7] Einige Tabellenkalkulationen – auch Microsoft Excel – erlauben zwar die Definition von prozeduralen oder objektorientierten Programmen in einer dafür geeigneten Sprache, dies erfolgt jedoch außerhalb des Formulars und beinhaltet einen Wechsel des Programmierparadigmas und der verwendeten Sprache.

Neben den Tabellenkalkulationen gibt es weitere Anwendungen, bei denen der formularorientierte Ansatz Verwendung findet, um dem Endanwender die Realisierung von neuer Funktionalität bzw. eine Anpassung der vorhandenen zu ermöglichen. In [KHB+05] wird beispielsweise ein formularbasiertes System zur Endnutzerprogrammierung im Bereich der webbasierten Systemadministration beschrieben. Die vorgestellte Anwendung A1 soll es Systemadministratoren erlauben, kleinere Tools zu erstellen und anzupassen.

Die formularbasierten Anwendungen bauen auf einem bereits erfolgreich eingesetzten und auch außerhalb akademischer Kreise sehr bekannten Programmierparadigma auf. Dies bietet den Vorteil einer verringerten Einarbeitungszeit und einer gewissen Grundvertrautheit der Anwender mit den Applikationen. Im Folgenden wird ein weiteres Programmierparadigma vorgestellt, welches ebenfalls Einsatz findet um Endnutzern die Erstellung von Anwendungen zu erleichtern, aber wesentlich weniger bekannt und verbreitet ist.

2.5.2 Programming by Example

Programmieren durch Beispiele[8] ist ein Konzept, welches neben der Endnutzerprogrammierung auch die Forschungsgebiete der künstlichen Intelligenz und der visuellen Programmierung berührt. Die Idee besteht darin, dass der Nutzer die zur Lösung einer Aufgabe nötigen Schritte dem Softwaresystem demonstriert und dieses dann in der Folge in der Lage ist, diese Aufgabe zu lösen.

In grundlegender Form ist dieser Ansatz in den so genannten Makrorekordern realisiert, hier kann ein Nutzer Interaktionen über Tastatur und Maus durchführen und die Makrorekordersoftware zeichnet diese Aktionen im Hintergrund auf. Die gesamte Folge der Interaktionen kann dann beim nächsten Mal abgespielt werden, eine unter Umständen komplexe Folge von Aktionen wird dadurch vereinfacht. Solche Werkzeuge finden sich beispielsweise in den Microsoft Office-Produkten.

Da hierbei nicht versucht wird, die Absichten des Nutzers zu interpretieren, sondern eine reine Wiederholung stattfindet, führen in der Regel schon geringfügige Änderungen dazu, dass solcherart erstellte Programme – oder im Fall der Makrorekorder Makros – nicht mehr anwendbar sind. Dementsprechend gibt es Ansätze, bei denen das Softwaresystem versucht, aus den vom Nutzer vorgeführten Beispielen zu abstrahieren und eine allgemeinere Lösung abzuleiten. Laut [Sch98] werden diese Ansätze in „Programmieren **mit** Beispielen“, darunter ist das reine Wiederholen zu verstehen, und „Programmieren **durch** Beispiele“ unterschieden. Letzteres berührt das Forschungsgebiet der künstlichen Intelligenz, da aus den durch den Anwender präsentierten Aktionen ein Lösungsweg für ähnliche Probleme abgeleitet werden muss.

Es existieren unterschiedliche Ansätze dafür, in welcher Form das Demonstrieren durch den Nutzer erfolgt. Frühe Ansätze arbeiteten mit so genannten Input-Output-Paaren, dabei versuchte das System eine geeignete Transformation zu finden. Spätere Systeme ließen den Nutzer die Aktionen auf Beispieldaten ausführen und versuchten diese Aktionen zu verallgemeinern.

Ein Problem besteht hierbei in der Schwierigkeit, zu garantieren, dass das Entwicklungswerkzeug die vom Anwender gewünschte Anwendung erzeugt [Sug01]. Dies hat zur Folge, dass unter Umständen eine Änderung des erzeugten Programms durch den Anwender erfolgen muss. Dies kann jedoch häufig – wenn es überhaupt möglich ist – nicht auf der ursprünglichen Interaktionsebene geschehen. Da die intern verwendete Repräsentation in der Regel von der durch den Nutzer manipulierten abweicht, entsteht ein semantischer Bruch. Der von den Programmiersystemen intern erzeugte Code ist häufig

[8] In der angelsächsischen Literatur sind sowohl die Begriffe **programming by example** als auch **programming by demonstration** üblich.

schwer verständlich und der ursprünglichen Repräsentation schwer zuzuordnen[9] [Sch98]. Dadurch gehen die Vorteile im Rahmen der Endnutzerprogrammierung verloren.

Eine vergleichbare Situation tritt auch auf, wenn ein Endnutzer versucht, die mit den eingangs angesprochenen Makrorekordern aufgezeichneten Abläufe anzupassen, um sie in ähnlichen Situationen wieder verwenden zu können. Um eine solche Verallgemeinerung vorzunehmen ist in der Regel eine Änderung der intern verwendeten Repräsentation notwendig. Am Beispiel des in Microsoft Word integrierten Makrorekorders wird deutlich, wie groß der dabei vom Endnutzer zu überbrückende semantische Bruch ist. Das in Abb. 2.5 dargestellte Programmfragment in Visual Basic ist die interne Repräsentation eines aufgezeichneten Makros. Die aufgezeichneten Aktionen bestanden im Markieren eines Textes, dann wurden die Schrifteigenschaften kursiv und fett eingestellt und der Absatz zentriert. Diese Aktionen werden mit der Maus ausgeführt, die gesamte Interaktion läuft wie bei der Bedienung von Windowsanwendungen üblich über das Anklicken von Schaltflächen.

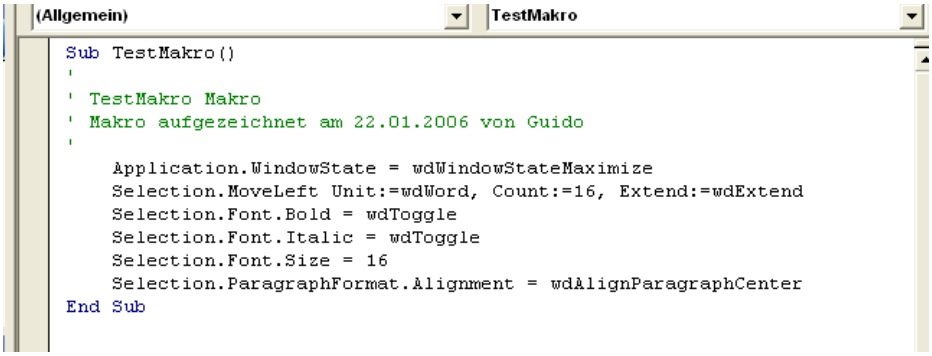


Abb. 2.5: Ein in Microsoft Word aufgezeichnetes Makro

Eine Anpassung des Visual Basic Codes setzt Kenntnisse dieser Sprache voraus und ist durch einen reinen Endanwender ohne zusätzliche Hilfestellung kaum zu leisten. Die inzwischen in einigen Entwicklungsumgebungen vorhandenen Ansätze zur Unterstützung des Anwenders durch das automatische Vervollständigen beziehungsweise Vorschlagen von Anweisungen, die im aktuellen Kontext sinnvoll sind, kann diese Aufgabe erleichtern. Aber auch mit diesen Hilfen ist ein gewisses Maß an Kenntnissen der verwendeten Programmiersprache unabdingbar.

In [Cyp94] sind eine ganze Reihe von Applikationen vorgestellt, die das Programmieren mit bzw. durch Beispiele umsetzen, beginnend bei Pygmalion – einer bereits 1975 von David C. Smith entwickelten Applikation. An dieser Stelle ist zu bemerken, dass trotz der vergleichsweise langen Forschungsgeschichte mit Ausnahme der oben angesprochenen Makrorekorder nur vereinzelt Anwendungen auf Basis dieses Programmierkonzeptes außerhalb des akademischen Bereiches Anwendung finden bzw. fanden. Ein Beispiel, welches den Sprung aus der akademischen Welt zu einem Produkt geschafft hat, ist das Programm Stagecast Creator[10].

Der Stagecast Creator zielt auf das Programmieren in einer grafischen Welt ab. Überwiegende Einsatzbereiche sind der Schul- und Unterrichtsbereich – hauptsächlich in unteren Klassenstufen – und die Entwicklung von einfachen Spielen auf dieser Plattform. Die unten stehende Abbildung zeigt das Programm, der Screenshot zeigt auf der linken Seite die so genannte „Stage“ (Bühne). Eine der Figuren ist ausgewählt und die linke Bildschirmhälfte zeigt das für diesen „Darsteller“ namens Buster

[9] Als Beispiel für diese Diskrepanz zwischen externer und interner Darstellung sei auf die Beschreibung des Programmiersystems Metamouse [MW94] verwiesen.

[10] Siehe [Sta06], dort ist auch eine Demoversion erhältlich.

angelegte Regelset. Ein Programm kann durch die Steuerknöpfe im unteren Bereich des Stagebildschirms gestartet, im Einzelschrittmodus ausgeführt und angehalten werden.

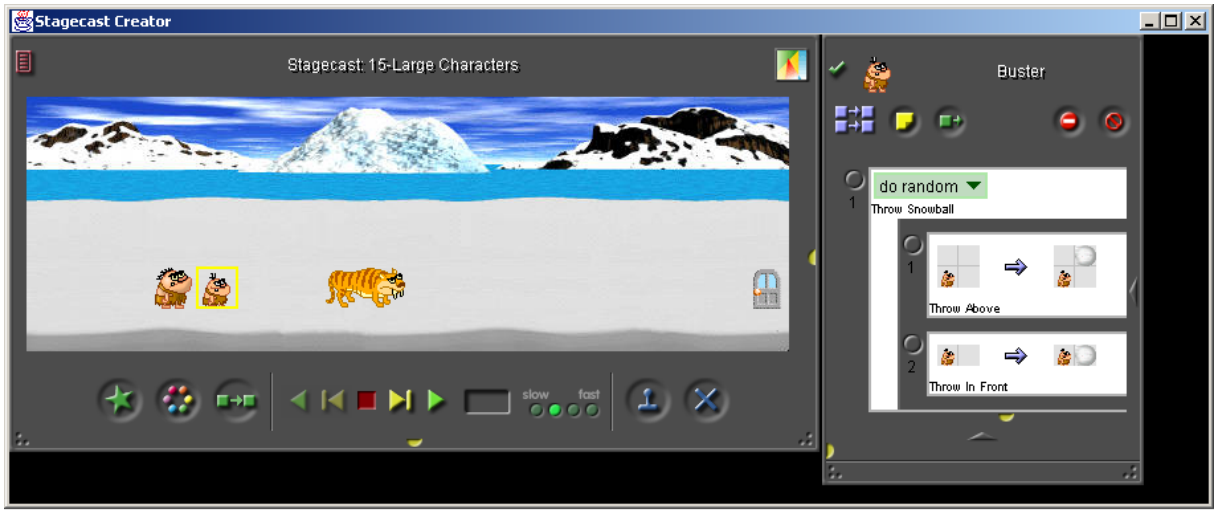


Abb. 2.6: Der Stagecast Creator

Das gesamte Erstellen von Regeln (Bedingung – Aktion) erfolgt über die Manipulation einer visuellen Repräsentation (siehe Abb. 2.7), daher ist der Stagecast Creator ein gutes Beispiel für die eingangs angesprochene Schnittmenge der Bereiche visuelle Programmierung und Programmieren durch Beispiele.

Wenn die Bedingung einer Regel erfüllt ist, in diesem Fall ein freies Feld vor der Figur, so wird die zugehörige Aktion ausgeführt, im Beispiel wird die Figur um ein Feld weiter bewegt. Programme werden derart ausgeführt, dass in einem Schritt die Regeln der Reihe nach auf Anwendbarkeit geprüft werden, die erste passende wird ausgeführt und es beginnt ein neuer Durchlauf. Wird in einem Programm die Regel aus dem Beispiel (Abb. 2.7) ausgeführt, dann bewegt sich der dargestellte Steinzeitmensch um ein Feld nach rechts.

Abb. 2.7: Das Erzeugen einer Regel

Der Stagecast Creator unterstützt einfache Tastatur- und Mausabfragen, darüber hinaus ist ein einfaches Variablenkonzept und grundlegende Arithmetik vorhanden. Insgesamt handelt es sich um eine sehr interessante Anwendung des Konzeptes Programmieren durch Beispiele in der

Endnutzerprogrammierung. Die gewählte Anwendungsdomäne ist gut geeignet um Anwendungen mit diesem Programmierparadigma erstellen zu können.

Neben der eben angesprochenen Anwendungsdomäne gibt es einige wenige weitere Bereiche in denen das Konzept des Programmierens durch Beispiele erfolgreich eingesetzt wurde, beispielsweise beschreiben [Sug01] eine Realisierung einer Anwendung zur Automatisierung von Webbrowsingaufgaben[11]. Dabei handelt es sich wie beim Stagecast Creator um eine eingeschränkte Anwendungsdomäne, innerhalb derer sich die Probleme der Realisierung von Anwendungen für das Programmieren durch Beispiele vergleichsweise gut beherrschen lassen. Eine allgemeine Lösung ist schwierig (siehe ebenfalls [Sug01]), so dass dieses interessante Paradigma wohl auch in Zukunft auf einzelne, eng umrissene Anwendungen beschränkt bleiben wird.

2.6 Fazit und Schlussfolgerungen für die vorliegende Arbeit

Das Thema Endnutzerprogrammierung ist ein wichtiges Forschungsgebiet für die angewandte Informatik. Zum einen bietet der aktuelle Stand der im Einsatz befindlichen Konzepte und der darauf basierenden Werkzeuge noch Raum für Verbesserungen. Zum anderen hat dieses Forschungsgebiet auch eine nicht unerhebliche ökonomische Bedeutung. Die Aufgaben, welche mit Unterstützung von Softwareanwendungen gelöst werden, nehmen weiter zu. Mit dem verstärkten Einsatz von Anwendungsprogrammen treten aber auch immer häufiger Situationen auf, in denen eine Standardsoftware „aus dem Regal“ mehr oder weniger großer Anpassungen bedarf, um für eine Aufgabe geeignet zu sein. Oder es entsteht der Bedarf nach Softwarelösungen, die nicht am Markt verfügbar, durch einen Domänenexperten aber rasch umsetzbar wären.

Aufgrund der Ausführungen in diesem Kapitel ergeben sich eine Reihe von Anforderungen an ein Werkzeug zur Endnutzerprogrammierung. Der Lernaufwand muss im Vergleich zu den etablierten Programmiersprachen, mit denen Softwareentwickler ihre Probleme lösen, deutlich gesenkt werden. Zu diesem Zweck ist eine Erhöhung des Abstraktionsniveaus notwendig; das Ziel muss sein, dass ein Anwender eine Lösung in für ihn verständlichen Begrifflichkeiten formulieren kann. Im weiteren Verlauf dieser Arbeit werden verschiedene Ansätze untersucht, mit denen zu diesem Ziel beigetragen werden kann. Berücksichtigt werden Komponenten, domänenspezifische Sprachen und Generatoren.

Die im Abschnitt über das „Programmieren durch Beispiele“ erwähnte semantische Lücke kann immer auftreten, wenn mit unterschiedlichen Präsentationen zur Lösungsbeschreibung interagiert werden soll. Wünschenswert wäre an dieser Stelle eine möglichst schmale semantische Lücke. Da in dieser Arbeit ein Konzept entworfen wird, wie Endnutzerprogrammierung durch unterschiedliche Darstellungsformen des Lösungsmodells unterstützt werden kann, ist dieser Punkt bei den weiteren Betrachtungen zu beachten. Dabei sollen grafische und textuelle Notationen zum Einsatz kommen, um unterschiedliche Phasen des Entwurfs der Anwendungsrealisierung zu unterstützen.

In diesem Kapitel wurden Beispiele von Einsatzbereichen für Endnutzerprogrammierung angesprochen, in denen sich dieses Konzept vorteilhaft und mit vertretbaren Risiken nutzen lässt. Im weiteren Verlauf der Arbeit werden neben den konzeptuellen Grundlagen und der Realisierung eines prototypischen Endnutzerprogrammiersystems auch mögliche Nutzungsszenarien für diese Anwendung beschrieben. Ein interessanter Aspekt der Realisierung von Softwareapplikationen für die Endnutzerprogrammierung ist der Umstand, dass der Entwickler eines solchen Systems zum Meta-Designer wird. Im Allgemeinen ist ein Endnutzerprogrammiersystem hinsichtlich der möglichen

[11] Die beschriebene Anwendung „Scrapbook“ soll auch kommerziell vermarktet worden sein, dies konnte jedoch nicht verifiziert werden.

Anwendungen, die sich damit entwickeln lassen, eingeschränkt. Der Entwurf eines solchen Programmiersystems bezieht sich also nicht nur auf die eigentliche Entwicklungsanwendung, sondern beschreibt eine Familie realisierbarer Anwendungen im Sinne einer möglichen Lösungsmenge für einen eingeschränkten Problemraum.

Die in dieser Arbeit unter diesem Gesichtspunkt entwickelten Ansätze verwenden in hohem Maße so genannte Komponenten. Daher wird dieses Konzept im folgenden Kapitel eingehend diskutiert.

3 Softwarekomponenten

„Adde parvum parvo magnus acervus erit."
(Füge wenig zu wenig und das Ergebnis wird ein großer Haufen sein)
Ovid, römischer Dichter, 43 v. Chr. - 17 n. Chr.

Im vorangegangenen Kapitel wurde eine Einführung in den Themenbereich der Endnutzerprogrammierung gegeben. Ein Weg, um die Erstellung von Software durch Nichtentwickler zu ermöglichen, ist das Komponentenkonzept. Häufig ist zur Verdeutlichung dieser Idee die Legostein-Metapher zu finden. Ähnlich den Gebäuden und Modellen bei diesem Kinderspielzeug soll eine Softwareapplikation aus einzelnen Bestandteilen zusammengesetzt werden. Idealerweise ist auch bei einer fertigen Anwendung der Austausch einzelner Bausteine gegen andere Exemplare genauso einfach.

Zunächst wird motiviert, warum objektorientierte Konzepte alleine nicht ausreichend sind, um die Anforderungen, die sich aus diesem Ziel ergeben, zu erfüllen. Insbesondere hinsichtlich Wiederverwendung und Wartbarkeit zeigt sich der Bedarf nach weiter gehenden Ansätzen.

Bei der Auseinandersetzung mit dem Thema Komponenten zeigte sich schnell, dass keine einheitliche Definition in der Literatur existiert. Daher wird auf Basis der in der Literatur vorhandenen Definitionen eine Definition für die vorliegende Arbeit abgeleitet. Die Eigenschaften des Komponentenbegriffs werden zunächst nach den Aspekten Strukturierungsmittel, Verteilungseinheit und Baustein gegliedert betrachtet und es werden entsprechende Teildefinitionen abgeleitet. Diese werden am Ende der Betrachtungen zusammengefasst.

Den Abschluss des Kapitels bildet eine kurze Gegenüberstellung des Komponentengedankens und des Service-Konzeptes, um den Zusammenhang zwischen beiden aufzuzeigen.

3.1 Einleitung und Motivation

Zu den Erwartungen, die von Beginn an mit dem Paradigma der Objektorientierung (vgl. [JFF+02], [Amb01]) verbunden waren, gehören insbesondere erhöhte Wiederverwendbarkeit, leichtere Erweiterbarkeit und bessere Wartbarkeit der entwickelten Programme. Es ist nicht unumstritten, ob diese Versprechen eingelöst werden können[12]. Im Folgenden werden einige Punkte diskutiert, an denen die durch die Objektorientierung zur Verfügung gestellten Mittel die Probleme nicht optimal lösen und zusätzliche Konzepte erforderlich sind, um diese Ziele zu erreichen.

Der folgende Abschnitt soll dabei keinesfalls als Kritik an der Objektorientierung im Allgemeinen aufgefasst werden, vielmehr stellt die objektorientierte Programmierung nach meiner Ansicht einen wichtigen Schritt in der Softwareentwicklung dar. Die objektorientierte Softwareentwicklung hat die Komplexität großer Softwaresysteme wesentlich besser beherrschbar gemacht, als dies auf Basis der zuvor verfügbaren Ansätze möglich war. Allerdings ist sie als Strukturierungs- und Beschreibungsmittel auf einer Ebene unterhalb der modularen Zerlegung eines Gesamtsystems anzusiedeln. Die objektorientierte Programmierung stellt ein mächtiges und ausgereiftes Werkzeug für die Programmierung innerhalb größerer Bausteine eines Gesamtsystems dar; auf der Ebene der Komposition des Gesamtsystems und der Interaktion dieser Einzelteile sind jedoch weiter gehende Konzepte notwendig. Die auf Parnas [Par72a] [Par72b] zurückzuführenden Ansätze zur Strukturierung von Software auf Basis von Modulen zeigen bereits vergleichbare Anforderungen.

[12] Teilweise wird die Diskussion hierüber sehr emotional geführt [Fin05], eine Betrachtung aus Entwicklersicht findet sich unter [Cli03]

Objektorientierte Softwareentwicklung kann – wie auch die anderen Ansätze der letzten Jahre – kein Allheilmittel für alle Probleme der Softwareentwicklung sein; nichtsdestoweniger war in den letzten Jahren in vielen Bereichen eine gewisse Fixierung auf die vermeintliche „Silberkugel“ zu beobachten[13]. Die grundlegende Idee der Objektorientierung – ein Objekt stellt eine Kapselung von Daten und Funktionalität dar – ist nicht immer eine geeignete Abstraktion der Wirklichkeit. Zwanghaftes Festhalten an diesem Paradigma führt in einigen Anwendungsszenarien zu einem unbefriedigenden Modell. Ein Beispiel für solche Schwierigkeiten bei der objektorientierten Modellierung ist die Abbildung von Graphstrukturen oder relationalen Strukturen auf ein objektorientiertes Modell. Im Falle einer symmetrischen Interaktion zwischen zwei Klassen ergibt sich vielfach die Notwendigkeit, die entsprechende Funktionalität – beispielsweise zum Transfer von Informationen – in einer der beiden Klassen anzusiedeln oder eine weitere, nur durch diese Interaktion motivierte, Klasse in das Modell einzuführen.

Die Granularität von Objekten ist häufig zu fein, um eine geeignete Basis für die Wiederverwendung zu bieten (vgl. [BS02]). Dies liegt darin begründet, dass ein Objekt selten in der Lage ist, die von ihm bereitgestellten Dienste ohne Verwendung anderer Objekte zu erbringen; die hieraus resultierende Kontextabhängigkeit behindert die Wiederverwendung (vgl. auch [Sma99], [GT00]).

Bei der Wiederverwendung eines Objekts oder genauer der zugehörigen Klasse im Rahmen einer neuen Anwendung müssen in der Folge alle Objekte anderer Klassen identifiziert werden, welche dieses für die Erbringung der benötigten Funktionalität benötigt. Diese Klassen müssen ebenfalls übernommen und die relevanten Beziehungen beibehalten werden. Dieses Vorgehen muss entsprechend für die übernommenen „Hilfsobjekte“ iteriert werden, so dass man sich häufig gezwungen sieht, wesentlich größere Teile des Objektgeflechtes zu übernehmen als ursprünglich angedacht. Der Schritt, die an einer Kollaboration beteiligten Objekte zu identifizieren, wird außerdem mit steigender Größe und Komplexität eines Softwaresystems schwieriger, oder anders gesagt „den Steuerfluss einer objektorientierten Anwendung zu verfolgen, sei wie eine Landkarte durch einen Strohhalm zu betrachten.“ [JH02].

Da eine Kapselung auf der Ebene der Klassen die Wiederverwendung nicht in geeigneter Weise unterstützt, sondern Modularität auf der Ebene eines Verbundes der miteinander interagierenden Objekte nötig ist, bieten viele der heute gängigen Programmiersprachen Gruppierungsmöglichkeiten für Klassen ähnlich dem Modulkonzept der strukturierten Programmierung. Diese Ansätze beschränken sich jedoch häufig auf ein reines logisches Gruppieren und bieten nur unzureichende oder gar keine Unterstützung für Versionierung, gemeinsames Interface für die kollaborierenden Klassen oder Umsetzung des Black-Box-Prinzips.

Die eben angesprochene hohe Komplexität der Wiederverwendung auf Klassenebene macht es unmöglich, Software aus diesen Bausteinen durch „Nicht-Programmierer“ entwickeln zu lassen [SGM03]. Im Rahmen der Endnutzerprogrammierung kann dieser Ansatz also nicht in dieser Form zum Einsatz kommen. Eine Zusammenfassung der interagierenden Klassen in einer geeigneten Form ist nötig.

Darüber hinaus ist es in den seltensten Fällen möglich, eine Klasse in genau der Form, wie sie ursprünglich entworfen und implementiert wurde [Ric99], innerhalb einer Anwendung wieder zu verwenden. Im Allgemeinen ist hierfür eine Anpassung erforderlich. Die Objektorientierung unterstützt Anpassung durch Aggregation – also das Erstellen einer umhüllenden Klasse – oder durch das Erstellen einer angepassten Klasse durch die Verwendung des Vererbungskonzeptes. Letzteres ermöglicht die Anpassung von Klassen, indem von allgemeineren Klassen spezielle abgeleitet werden

[13] Ähnliches wiederholt sich mit Komponenten und Services.

können und erhöht damit auf den ersten Blick die Wiederverwendbarkeit. Allerdings fördert dieses Konzept die Entstehung schwer zu durchschauender impliziter Abhängigkeiten der Klassen untereinander, welche wiederum die Wiederverwendung erschweren und den Prozess fehleranfälliger machen. Zusätzlich zu den oben erwähnten „horizontalen“ Abhängigkeiten zwischen den Klassen eines Systems aufgrund der Interaktion ihrer Instanzen, entsteht hier eine weitere „vertikale“ Dimension der Abhängigkeit aufgrund der Vererbungshierarchie. In Programmiersprachen mit Unterstützung für Mehrfachvererbung wird dieses Problem durch die rasch steigende Komplexität innerhalb der Vererbungshierarchie zusätzlich verschärft. Ein Verzicht auf dieses Konzept verkompliziert die Wiederverwendung in den Fällen, in denen eine Klasse in eine bestehende Klassenhierarchie eingefügt werden muss.

Vererbung wird häufig im Sinne von Implementationsvererbung verwendet; eine Unterklasse erbt nicht nur das Interface respektive das Verhalten einer Oberklasse sondern ihre Implementation. Dies führt dazu, dass eine Änderung an der Oberklasse eventuell die Implementation der Unterklassen „zerbrechen“ kann. Dieses Problem ist unter dem Begriff **fragile base class problem** Gegenstand zahlreicher Veröffentlichungen und trägt ebenfalls dazu bei, den Wiederverwendungsprozess zu erschweren. Eine detaillierte Betrachtung ist in [MS98] zu finden. Ansätze wie **Design by Contract** (siehe z. B. [Mey97]) können dazu beitragen, solcher Probleme schneller gewahr zu werden. Allerdings sind diese Konzepte nicht in allen objektorientierten Sprachen[14] realisiert und stellen darüber hinaus keine Lösung für das ursächliche Problem dar.

Eine geeignete Modularisierung erhöht jedoch nicht nur die Wiederverwendbarkeit einzelner Bausteine eines Softwaresystems, sie hat darüber hinaus auch Einfluss auf die Wartbarkeit eines Softwaresystems. Stellen die Grenzen eines solchen Bausteines gleichzeitig Grenzen für die Abhängigkeiten innerhalb des Softwaresystems dar, verfügt ein Baustein also nur über explizite Abhängigkeiten zu seiner Umgebung, so zieht eine Änderung an diesem keine Veränderungen an anderen Bausteinen nach sich. Dieses Prinzip ist unter dem Begriff **lokale Änderbarkeit** seit langem im Software Engineering bekannt. Die hohe Anzahl an Veröffentlichungen zu unterschiedlichen Konzepten, die mit dem Ziel der Modularisierung von Software in den letzten Jahren entwickelt wurden (Aspektorientierte Programmierung (AOP), Mixins [Sma99], Komponentenorientierung, Services) zeigt, dass die durch die Objektorientierung bereitgestellten Möglichkeiten zur Modularisierung und Komposition großer Systeme nicht ausreichend sind. Eine intensive Untersuchung der Möglichkeiten im Bereich objektorientierter Komposition findet sich bei [OM01]; auch dieser Autor kommt zu dem Schluss, dass die „traditional object-oriented composition mechanisms“[15] in einigen Anwendungsszenarien nicht befriedigend sind.

Die zwischen den Instanzen der an einem Anwendungssystem beteiligten Klassen zur Laufzeit bestehenden Verbindungen werden zum größten Teil bereits zur Entwicklungszeit in Form von Methodenaufrufen festgelegt [Gri01]. Ausnahmen hiervon sind nur insoweit möglich, dass lediglich die Signatur einer Methode festgelegt ist und die konkrete Implementierung dieser Methode zur Laufzeit in einer Unterklasse oder einer dieses Interface implementierenden Klasse gesucht wird. Eine Änderung der Methodensignatur oder beispielsweise der Abfolge der Aufrufe ist zur Laufzeit im Allgemeinen nicht mehr möglich. Dadurch wird der zur Laufzeit existierende Objektgraph – und damit die möglichen Interaktionen zwischen den beteiligten Objekten – in den gängigen Programmier-

[14] In den zurzeit verbreiteten objektorientierten Entwicklungssprachen C++, Java, C# und VB.NET sind diese Ansätze nicht vorhanden. Mit Eiffel [Eif06] steht eine Sprache zur Verfügung, die kompatibel zur Common Language Specification des .NET-Frameworks – näheres hierzu findet sich in Kapitel 5 – ist und das Konzept des Design by Contract vollständig umsetzt.

[15] Gemeint sind hier Vererbung und Komposition.

sprachen bereits zur Entwurfszeit festgelegt. Der ursprünglichen Idee der Objektorientierung, eine Applikation als ein System miteinander kommunizierender Objekte zu betrachten, ist also ein vergleichsweise enger Rahmen gesetzt. Dieser Rahmen beschneidet auch die aus der Realität gewonnene Anschauung von Interaktion; hier ist zwar ein Kennen der beteiligten Akteure untereinander notwendig, die Kollaborationen sind aber über die „Lebenszeit“ eines Akteurs hinweg wesentlich flexibler.

Diese Punkte motivierten zahlreiche Entwicklungen, eine von diesen sind **Softwarekomponenten**. Die Idee, Software aus größeren Bestandteilen und damit auf einer höheren Abstraktionsebene zusammenzubauen, verspricht einige Vorteile. Diese reichen von schnelleren Entwicklungszeiten über Wiederverwendung bewährter Bausteine bis hin zur Komposition der Bausteine durch Nicht-entwickler, in diesem Zusammenhang ist das Konzept auch im Rahmen der vorliegenden Arbeit von Interesse. Da in der Literatur keine allgemein anerkannte Definition des Begriffes Komponente bzw. Softwarekomponente existiert – Appendix B bietet einen kleinen Überblick über das Spektrum der zu findenden Definitionen – wird im Folgenden zunächst eine für diese Arbeit zu verwendende Definition gegeben.

3.2 Der Komponentenbegriff

> *At a recent conference called "Object and Component Stuff", I stopped a posse of experts passing in the hallway and accosted them with a simple request: "Er, excuse me, could you tell me what a component is, please?" With a hurried backward glance, many of them scurried quickly away, as nervous as if I'd panhandled them for some spare airline upgrade coupons.*[16]
>
> *Meilir Page-Jones in Fundamentals of Object-Oriented Design in UML*

Der Begriff Komponente oder genauer Softwarekomponente ist – wie Meilir Page Jones in seinem Buch anekdotenhaft andeutet – schwer zu fassen und in der Literatur nicht genau umrissen. Vielfach existiert eine Vorstellung davon, was unter einer Komponente zu verstehen sei, doch selbst diese Vorstellungen klaffen weit auseinander. Dabei werden Komponenten zum einen auf sehr unterschiedlichen Granularitätsebenen betrachtet, zum anderen werden je nach Autor der Definition verschiedene Aspekte in der Definition berücksichtigt.

Der Begriff Komponente oder Komponentenorientierung ist nicht neu, bereits auf der NATO Softwarekonferenz 1968 – hier wurde auch der Begriff des Software Engineering geprägt – wurde dieser Begriff von M.D. McIlroy verwendet [McI68]. Zu diesem Zeitpunkt wurden jedoch Routinen als Softwarekomponenten bezeichnet. Es wurde also eine Gliederung der Software auf der Ebene von Prozeduren und Funktionen betrachtet. Seit dieser Zeit wurde der Begriff Komponente immer wieder für „Softwarebausteine“[17] auf unterschiedlichen Abstraktionsebenen verwandt. Vor diesem Hintergrund erscheint es verständlich, dass viele ein intuitives Verständnis davon haben, was eine

[16] „Auf einer kürzlichen Konferenz mit Namen „Objekt- und Komponentenzeug“, stoppte ich bei einer Gruppe von Experten, die durch den Flur gingen und sprach sie mit einer einfachen Frage an: „ Äh, entschuldigen Sie, könnten Sie mir bitte sagen, was eine Komponente ist?“ Mit einem eiligen Blick zurück hasteten viele von ihnen schnell weg, so nervös, als hätte ich sie um ein paar überflüssige Gutscheine für ein Flugupgrade angebettelt.“

[17] Die so genannte Legostein-Metapher wird häufig herangezogen, um den Begriff Softwarekomponente zu verdeutlichen. Diese zeigt sehr schön die intuitive Vorstellung, dass eine Komponente Baustein eines „größeren“ Softwaresystems ist, also mit anderen Komponenten zusammengesetzt werden kann um etwas Größeres zu formen.

Komponente sein soll. Durch die lange Verwendung in unterschiedlichen Kontexten fällt eine präzise Definition indes schwer.

Die in der Literatur zu findenden Definitionen reichen von mathematischen Ansätzen (siehe [Bro96] [Bro98]) über den Versuch, eine Komponente als abstraktes Konzept des Software Engineering zu beschreiben, bis hin zu sehr technischen und realisierungsabhängigen Beschreibungen in den verschiedenen Komponentenmodellen (COM, CORBA,...). Die zu findenden Definitionen stellen dabei häufig eine Zusammenfassung der im jeweiligen Anwendungsszenario wünschenswerten Eigenschaften oder der gewünschten Verwendungsmöglichkeiten der Komponenten dar.

Der Komponentenbegriff sollte keine Einschränkung hinsichtlich einer konkreten Realisierung innerhalb eines Programmierparadigmas implizieren; es sollte unerheblich sein, ob eine Komponente intern mittels prozeduraler, objektorientierter oder anderer Konzepte umgesetzt wird.

Bei dem Versuch, eine Definition für den Begriff Komponente zu geben, sollte die Unterscheidung zwischen Komponenten und verteilten Systemen beachtet werden. Diese beiden Problembereiche liegen orthogonal zueinander, auch wenn vielfach verteilte Systeme auf der Basis von Komponententechnologien realisiert werden. Für die Begrifflichkeit sollte es aber unerheblich sein, in welchem Kontext eine Komponente Anwendung findet.

Ursprünglich sollte eine kurze Definition des Komponentenbegriffs für die Verwendung in dieser Arbeit gegeben werden. Da sich hierbei – wie oben bereits angesprochen – sehr unterschiedliche Aspekte der Betrachtung ergaben und eine Vielzahl unterschiedlicher Definitionen gefunden wurde, erfolgt eine Diskussion des Begriffes. Die Herleitung der Definition wird nach mehreren Gesichtspunkten gegliedert vorgenommen und am Ende zusammengefasst.

3.2.1 Komponente als Strukturierungsmittel

Die Zusammenfassung mehrerer kleinerer Teile zu einem größeren Ganzen ist unter dem Begriff Kapselung im Software Engineering ein bereits erprobtes Konzept zur Reduktion von Komplexität (siehe z. B. [PB96]). Dabei werden ein möglichst großer inhaltlicher Zusammenhalt und eine möglichst geringe Abhängigkeit vom umgebenden System angestrebt. Eine Komponente stellt eine Realisierung dieses Kapselungsprinzips dar. Sie repräsentiert ein logisch zusammenhängendes Ganzes und grenzt dieses gegen die Umwelt ab.

Eine Komponente kann ebenfalls eine Zusammenfassung anderer, logisch zusammengehöriger Entitäten sein [AR03], [DW99]. Es sind aber auch monolithische Komponenten denkbar, die nicht weiter unterteilbar sind. Ein Beispiel für Komponenten als Zusammenfassung von Entitäten auf einem niedrigeren Abstraktionslevel sind mehrere Klassen, die gemeinsam benötigt werden um eine Funktionalität zu realisieren. Eine mit prozeduralen Sprachkonzepten entwickelte Komponente kann eine Sammlung mehrerer Funktionen sein, die in einem Anwendungszusammenhang stehen. Es lassen sich viele derartige Beispiele finden, bei denen die zusammengefassten Entitäten unterschiedlichster Natur sind. Für die Definition des Komponentenbegriffs spielt die Art bzw. Granularität jedoch keine Rolle, da eine Komponente in ihrer Verwendung konzeptionell als unteilbare Einheit angesehen wird. Die unter dieser Abstraktionsebene anzusiedelnden Konzepte oder Realisierungen sind zu vielfältig, als dass man sie in einer sinnvollen Definition berücksichtigen könnte.

Unter dem Gesichtspunkt der angestrebten Definition sind die starke inhaltliche Kohäsion und die Abgrenzung die zentralen Punkte. Festzuhalten ist also:

Eine Komponente realisiert das Kapselungsprinzip. Ziele beim Entwurf einer Komponente sind eine starke inhaltliche Kohäsion und möglichst geringe Kopplung zu ihrem Umfeld.

Unter diesem Gesichtspunkt gibt es keine Unterscheidung zum Begriff des Moduls. Eine Reduktion des Komponentenbegriffs rein auf die Funktion als Strukturierungsmittel („*Verpackungseinheiten von Klassen*“ [GP98]) ist jedoch eine zu starke Vereinfachung.

Die inhaltliche Kohäsion und die geringe Kopplung zum Umfeld sind Eigenschaften, welche durch Entscheidungen beim Entwurf der Komponente bestimmt werden. Komponententechnologien ermöglichen es, diese Ziele beim Entwurf umzusetzen, können dies in der Regel aber nicht erzwingen. So lässt sich auf Basis einer Komponententechnologie sowohl eine Komponente entwerfen, welche völlig unzusammenhängende Funktionalitäten zusammenfasst, als auch eine Komponente, welche so unvollständig ist, dass eine Vielzahl von Beziehungen zum Kontext notwendig sind. Beide Beispiele entsprechen nicht der Idealvorstellung einer Komponente. Aus diesem Grund sind die Entwurfsziele in der Definition enthalten.

Neben diesen Entwurfsanforderungen an Komponenten gibt es Anforderungen, die mehr auf die technische Realisierung abzielen. Die im nächsten Abschnitt diskutierte Eignung einer Komponente als Baustein größerer Systeme fällt in diese Kategorie.

3.2.2 Komponente als Baustein

Eine Komponente soll als Baustein für größere Systeme geeignet sein. Komponenten sollen also mit anderen Komponenten interagieren können. Um die Nutzung einer Komponente zu ermöglichen, sind entsprechende Schnittstellen – diese müssen nicht dem aus objektorientierten Programmiersprachen bekannten Schnittstellenkonzept entsprechen – notwendig. Im Folgenden wird, wenn nötig, der Begriff **Komponentenschnittstelle** verwendet, um diese konzeptionellen Schnittstellen zur Umgebung einer Komponente von anderen Schnittstellenkonzepten, die unter Umständen in der Realisierung eine Rolle spielen, abzugrenzen. Die Forderung nach Schnittstellen als Bestandteil der Komponentendefinition ist auch in vielen in der Literatur vorhandenen Definitionen vorhanden [Ous97] [SGM03] [AR03] [RJB99] [BSW02]. Die explizite Festlegung von Schnittstellen soll die Austauschbarkeit einer Komponente oder eine Änderung der internen Realisierung ermöglichen, ohne dass eine Anpassung der Nutzer der Komponente erforderlich wird, solange die Schnittstelle sich nicht verändert. Die Verwendung von Schnittstellen für diese Zwecke ist in der Softwareentwicklung weit verbreitet, für weitere Ausführungen zu diesem Thema sei beispielsweise auf [Som01] verwiesen.

Die Forderung nach definierten Schnittstellen zum Zugriff auf die von einer Komponente bereitgestellte Funktionalität ist weit verbreitet, so spricht beispielsweise [Rei00] von expliziten Schnittstellen zu dem von der Komponente angebotenen Dienst. Um den Komponentenbegriff nicht auf monolithische Komponenten zu beschränken, welche die gesamte zur Bereitstellung ihres Angebotes benötigte Funktionalität selbst erbringen müssen, ist über die Betrachtung von reinen „Zugriffsschnittstellen“ hinaus die Betrachtung der durch die Komponente benötigten Schnittstellen geboten. Über diese kann eine Komponente beispielsweise auf Infrastrukturdienste zugreifen (vgl. [CSP02] [AR03], [NL97]). Um diese Unterscheidung, wenn nötig, verdeutlichen zu können, werden die Begriffe **Import- und Exportschnittstellen** – jeweils aus Sicht der Komponente – verwendet.

Diese Forderung nach expliziten Komponentenschnittstellen wird in [GP98] [SGM03] um die Forderung ergänzt, dass eine Komponente nur „explizite Abhängigkeiten zu ihrer Umgebung“ besitzt. Dies ist insbesondere im Hinblick auf die Eignung zur Wiederverwendung und zur Komposition ohne Kenntnis der internen Strukturen oder des Quellcodes (vgl. [GP98]) eine wichtige Eigenschaft.

Undokumentierte oder nicht erkennbare Abhängigkeiten und Seiteneffekte machen eine solche Verwendung unmöglich oder erschweren sie zumindest unnötig.

[BRS+98] unterscheidet drei Typen von Schnittstellen, neben **export interfaces** und **import interfaces** – diese entsprechen den zuvor angesprochene Im- bzw. Exportschnittstellen – werden **combined interfaces** als Schnittstellen mit den kombinierten Eigenschaften der beiden vorangegangenen angeführt. Eine solche Mischung der Schnittstellen scheint durch Implementationsgegebenheiten motiviert und sollte nach meiner Ansicht nicht in die konzeptionelle Betrachtung einfließen. Auch die Anzahl der Komponenteninterfaces sollte in der Definition keine Erwähnung finden (siehe z. B. den Vorschlag Koskimies in [BDH+98]). Bei einer Reduktion auf den Gesichtspunkt des Im- bzw. Exports von Funktionalität durch die Komponente können einzelne Schnittstellen immer zusammengefasst werden. Umgekehrt ist es durch eine Absenkung des Abstraktionslevels auch möglich, eine Schnittstelle als Komposition mehrerer einzelner Schnittstellen zu sehen und diese unabhängig voneinander zu betrachten.

Im Rahmen der hier entworfenen Definition wird keine Vorgabe hinsichtlich der Form der Spezifikation der Schnittstellen oder ihrer Realisierung gemacht. Prinzipiell lassen sich syntaktische und semantische Schnittstellenbeschreibungen unterscheiden. In der Regel wird zumindest erstere vorhanden sein müssen, um eine Nutzung einer Komponente zu ermöglichen. Für syntaktische Schnittstellenspezifikation existieren in der Praxis unterschiedliche Realisierungsformen in den vorhandenen Komponentenmodellen. Diese reichen von der Verwendung des objektorientierten Interfacekonzeptes, wie es beispielsweise im .NET-Framework üblich ist, bis zu sprachunabhängigen Beschreibungssprachen für Interfaces (Interface Description Language, IDL), wie man sie in CORBA findet. Für eine semantische Schnittstellenbeschreibung existieren zurzeit keine derartig weit verbreiteten Ansätze.

Die Verwendung einer Komponente sollte ohne Kenntnisse der inneren Strukturen bzw. des Quelltextes möglich sein. Eine Komponente stellt unter dem Gesichtspunkt der Wiederverwendung eine Black-Box dar (vgl. auch [NL97] [Bro96]), diese Betrachtungsweise stärkt die unter 3.2.1 besprochene Eigenschaft einer Komponente als Realisierung des Kapselungsprinzips.

Für die Definition folgt daraus:

> *Eine Komponente besitzt nur explizite Kontextabhängigkeiten. Sie besitzt Schnittstellen zum Zugriff auf die von ihr bereitgestellte Funktionalität und zum Import von Funktionalität, die sie benötigt. Die Verwendung einer Komponente ist ohne Kenntnis ihrer internen Realisierung möglich.*

Im Vergleich zum objektorientierten Ansatz, der in Abschnitt 3.1 diskutiert wurde, ist zu bemerken, dass eine Komponente den in ihrem Anwendungszusammenhang wesentlichen Teil der für die Bereitstellung ihrer Funktionalität benötigen Dienste beinhaltet. Ausnahmen hiervon können zum Beispiel allgemeine Infrastrukturdienste sein, welche durch die Ausführungsumgebung bereitgestellt werden. Im Gegensatz zur angesprochenen Situation bei der Wiederverwendung einer Klasse bestehen also weniger – und nur explizit beschriebene – Kontextabhängigkeiten, es kommt nicht zum Effekt des „Nachziehens“.

3.2.3 Komponente als Verteilungseinheit

Eine wichtige Anforderung an Komponenten ist, dass diese verteilbare[18] Einheiten darstellen und darüber hinaus diese Verteilung weitgehend unabhängig von anderen erfolgen kann [SVB03] [DW99] [Obj03a]. Die zugrunde liegende Idee besteht darin, dass eine Komponente ihre Funktionalität größtenteils selbst erbringen kann und möglichst nur Infrastrukturdienste, die durch die Ausführungsumgebung bereitgestellt werden, benötigt. Eine vollständige Unabhängigkeit von anderen Komponenten bei der Verteilung würde zu monolithischen Komponenten führen und die Auslagerung gemeinsamer Funktionalität unmöglich machen, dies ist nicht sinnvoll. Wenn eine Abhängigkeit von einer anderen Komponente besteht – eine Komponente zur Rechtschreibprüfung könnte beispielsweise eine Wörterbuchkomponente benötigen – so soll die Abhängigkeit auf das Interface beschränkt sein und nicht eine konkrete Realisierung erfordern.

An dieser Stelle stellt sich also die Frage nach einer sinnvollen Granularität. Diese Frage ist aber nur im Anwendungszusammenhang zu beantworten, die Komponente muss im angedachten Anwendungsszenario eine sinnvolle Größe haben.

[SGM03] fordert darüber hinaus, dass eine Komponente eine unteilbare Verteilungseinheit darstellt und niemals einzelne Teile einer Komponente verteilt werden. Dies ist im Hinblick auf die oben angesprochene Black-Box-Eigenschaft eine sinnvolle Forderung. Verteilung kann dabei aber nur im Sinne einer Weitergabe vom Entwickler an Nutzer meinen, die Wiederverwendung einzelner Teile einer Komponente – beispielsweise durch andere Komponentenentwickler – sollte nicht durch die Definition eingeschränkt sein.

Von der Eignung der Komponenten als Einheit unabhängiger Verteilung hängt es ab, ob sie in einem Markt gehandelt werden können. Teilweise werden in den zu findenden Definitionen Anforderungen an die Verwendbarkeit in einem Softwaremarkt direkt als Eigenschaften einer Komponente gefordert. Dies betrifft zum Beispiel die Eignung zur Komposition durch Dritte[19], in [SGM03] wird diese sogar als eine von drei charakteristischen Eigenschaften einer Komponente aufgeführt. Es finden sich auch Definitionen, die Komponenten explizit als "die Einheiten, die eingekauft und verwendet werden" [GP98] beschreiben. Vor dem Hintergrund der hier vorgenommenen konzeptuellen Betrachtung wird diese Perspektive nicht in die Definition aufgenommen. Eine detailliertere Betrachtung von Komponentenmärkten und damit verbundenen Fragen findet sich in [SGM03].

Eine Komponente sollte als ausführbare Einheit verteilt werden. Diese Forderung ist weniger stark als der häufig in der Literatur zu findende Begriff **binary unit** (siehe z. B. [SGM03], K.Koskimies und M.Stal in [BDH+98]). Eine Softwarekomponente wird in dieser Arbeit als ausführbar bezeichnet, wenn zu ihrer Verwendung kein vorhergehender Compilierungsschritt notwendig ist. Dabei ist die technische Realisierung, also ob die Softwarekomponente beispielsweise in Form eines Skriptes, als PE-Datei[20] auf einer Windows-Plattform oder als interpretierbarer Bytecode vorliegt, unerheblich. Gefordert wird, dass sie in der für sie vorgesehenen Ausführungsumgebung in der verteilten Form eingesetzt werden kann. Für die technische Realisierung ergibt sich hieraus die Anforderung, dass eine Interaktion von Komponenten möglich sein muss, die nur in ausführbarer Form vorliegen. Da es – wie oben erwähnt – ein Ziel ist, eine möglichst lose Kopplung von Komponenten mit ihrem Umfeld zu

[18] Im Komponentenumfeld ist häufig von Verteilung im Sinne verteilter Systeme die Rede, hier und im Folgenden ist jedoch die Verteilung im Sinne der Weitergabe von Software und der Softwareinstallation gemeint.

[19] Wie in [BDH+98] treffend bemerkt wird, bleibt bei dieser Definition offen, wer die ersten beiden Gruppen sind.

[20] Das PE-Dateiformat (Portable Executable) ist das in Windows verwendete Format für ausführbare Dateien. Für Einzelheiten zu diesem Format sei auf die Spezifikation [Mic99] verwiesen.

erreichen, sollte es in einer Ausführungsumgebung für Komponenten die Möglichkeit geben, Interaktionsbeziehungen für Komponenten nach der Übersetzungszeit noch ändern zu können[21].

Teilweise werden Forderungen hinsichtlich der Verwendbarkeit in die Definitionen mit aufgenommen. So findet sich beispielsweise in [Rei00] die Forderung nach einfacher Verwendbarkeit. Darunter wird die Möglichkeit zum Einsatz ohne aufwendige Konfiguration verstanden. Dies stellt eine Anforderung an den Verteilungsaspekt dar, welche allerdings nur schwer fassbar ist. So kann beispielsweise eine Komponente mit einer Vielzahl an einzelnen Konfigurationsparametern vergleichsweise leicht durch einen Nutzer in Betrieb genommen werden, wenn eine Unterstützung in Form eines Hilfsprogramms besteht. Es müsste also zwischen dem – individuell unterschiedlichen – Aufwand für einen Nutzer und einem eher technisch orientierten Aufwand, der beispielsweise die Anzahl der Parameter erfasst, unterschieden werden. Unabhängig von diesen Betrachtungen sollte auch eine aufwendig zu konfigurierende Komponente konzeptuell eine Komponente sein. Diese Eigenschaft wird daher nicht als Forderung im Rahmen der Definition berücksichtigt, eine möglichst einfache Verwendbarkeit sollte vielmehr als Entwurfsziel bei der Entwicklung von Komponenten bzw. von Software im Allgemeinen gesehen werden.

Die Ausführungen dieses Abschnittes führen zu der nachstehenden Definition:

> *Eine Softwarekomponente liegt in einer in der vorgesehenen Ausführungsumgebung ausführbaren Form vor. Sie besitzt eine in ihrem Anwendungsbereich sinnvolle Granularität und wird in diesem als unteilbar angesehen.*

Diese Definition ist gezwungenermaßen unscharf, da die Granularität einer Komponente nur vor dem Hintergrund eines konkreten Anwendungsbereichs definiert werden kann. Auch eine größere Anwendung kann in einem Szenario sinnvoll als eine einzelne unteilbare Komponente angesehen werden, während sie unter anderen Umständen eine Komposition mehrerer Komponenten darstellt. Als Beispiel hierfür mag ein Datenbanksystem im Einsatz bei einem Unternehmen dienen. Dies ist im Umfeld der anderen IT-Systeme in diesem Unternehmen eine einzelne, unteilbare Komponente. Für den Hersteller dieses Datenbanksystems hingegen besteht es aus einer Vielzahl einzelner Komponenten, welche getrennt weiterentwickelt und ausgetauscht werden können.

3.2.4 Weitere Betrachtungen

Über die bereits angesprochenen Punkte hinaus gibt es noch einige interessante Fragen im Zusammenhang mit der konzeptuellen Betrachtung von Komponenten, welche sich in keine der drei obigen Kategorien einordnen lassen.

Ähnlich wie in der Objektorientierung muss auch im Falle von Komponenten die Definition von der konkreten Instanz unterschieden werden. Häufig wird der Begriff Komponente synonym für beide Konzepte verwendet oder die Unterscheidung gar nicht getroffen. Nach meinem Erachten trägt dies aber zusätzlich zu den begrifflichen Schwierigkeiten bei. Im Folgenden werden zwei Punkte – Zustand und Identität – angesprochen, bei denen die Trennung der Konzepte wesentlich ist. Falls die Unterscheidung nicht aus dem Zusammenhang ersichtlich ist, werden im Rahmen dieser Arbeit die

[21] Diese Überlegungen sind dem Konzept der späten Bindung in der Objektorientierung ähnlich. In Abschnitt 4.3.2 werden Reflexion und dynamisches Laden von Komponenten angesprochen, auf Basis dieser Techniken lassen sich mit der späten Bindung vergleichbare Möglichkeiten auf Ebene der Komponenten umsetzen. Auch die im weiteren Verlauf dieser Arbeit angesprochenen Ansätze zur nachrichtenbasierten Komposition von Komponenten (siehe Kapitel 7 und 8) erlauben die Realisierung von Komponenteninteraktionen mit entsprechenden Eigenschaften.

Begriffe Komponenteninstanz und Komponententyp verwendet. Die bisherigen Ausführungen zum Komponentenbegriff beziehen sich in dieser Terminologie auf den Komponententyp.

Eng an diese Unterscheidung geknüpft ist die Frage nach dem Zustand einer Komponenteninstanz. In [SGM03] findet sich die Aussage, eine Komponente habe keinen von außen beobachtbaren Zustand[22], Kopien sollten sich von außen nicht voneinander unterscheiden lassen. In der Definition von [Spa00] findet sich die Ansicht, eine Komponente solle keinen Zustand behalten. In diesen Arbeiten wird nicht ausreichend zwischen Komponente und Komponenteninstanz unterschieden. Überlegungen zum Zustand sind nach meiner Ansicht Betrachtungen zur Komponenteninstanz, nicht zur Komponente[23] (im Sinne des Komponententyps). Die Ansicht, eine Komponenteninstanz solle keinen Zustand besitzen, wird im Rahmen der vorliegenden Arbeit nicht geteilt. Da eine Komponenteninstanz nach dem in dieser Arbeit vertretenen Verständnis beispielsweise innerhalb einer Gesamtaufgabe[24] mehrere aufeinander folgende einzelne Anfragen desselben Klienten bearbeiten kann, ist es wichtig, dass sie zwischen den einzelnen Anfragen einen Zustand behält. Dieser kann durch das Ansprechen der bereitgestellten Funktionalität verändert werden. Es ist denkbar, dass abhängig vom Zustand der Komponente Aufrufe zu unterschiedlichen Ergebnissen führen, damit wäre der Zustand beobachtbar.

Wenn Komponenten neben den bereits angesprochenen Import- und Exportschnittstellen auch eine Interaktionsschnittstelle für einen menschlichen Nutzer haben, so ist es absolut unumgänglich, einen Zustand zuzulassen und auch die Unterscheidbarkeit von Instanzen desselben Komponententyps zu fordern. Ein denkbares Szenario ist eine Komponente zur Anzeige von Informationen. In einem Gesamtsystem können mehrere Instanzen dieses Komponententyps vorkommen, alle Instanzen könnten dabei unterschiedliche Informationen präsentieren. Ein konkretes Beispiel hierfür ist die Realisierung eines elektronischen Warenkorbs im Rahmen einer internetbasierten Handelsplattform.

Es ist zweckmäßig, Komponenteninstanzen eine Identität zuzuordnen, ähnlich der Identität eines Objektes in der Objektorientierung. Dadurch sind auch zwei Komponenteninstanzen mit völlig identischem Zustand zwei separate Entitäten. Da in der Praxis Komponenten häufig objektorientiert entwickelt werden, also einen Verbund kooperierender Objekte mit jeweils eigener Identität darstellen, ist es sinnvoll, dass sich diese Eigenschaft auf die Komponenteninstanz überträgt.

Eine wünschenswerte Eigenschaft von Komponenten ist eine sprachunabhängige Verwendbarkeit. Hiermit ist gemeint, dass die Nutzer einer Komponente nicht notwendigerweise mit derselben Programmiersprache entwickelt worden sein müssen wie die Komponente selbst. Auch die Unabhängigkeit von einer konkreten Betriebssystemplattform ist anstrebenswert. Diese Eigenschaften sind interessant, da sie aus Komponentensicht den möglichen Einsatzbereich der Komponenten erweitern und aus Nutzersicht das Angebot der verfügbaren Komponenten erhöhen können. Beide Eigenschaften sind jedoch abhängig von der technischen Realisierung insbesondere der Ausführungsumgebung der Komponenten. Für die Begrifflichkeit spielen diese Eigenschaften hingegen keine Rolle. Sie sind deswegen auch nicht in der Definition berücksichtigt worden.

[22] S.36: „Finally, a component should not have any (externally) observable state – it is required that the component cannot be distinguished from copies of its own."

[23] Diese Betrachtung ist – wenn man den Fall statischer Attribute unberücksichtigt lässt – analog zu der in der Objektorientierung üblichen Sicht auf den Zusammenhang von Klassen, Objekten und deren Zustand.

[24] Dies kann beispielsweise eine Transaktion sein.

3.2.5 Zusammenfassung

Um eine geschlossene Darstellung zur Verfügung zu stellen, werden in diesem Abschnitt die zuvor diskutierten Eigenschaften des Komponentenbegriffs kurz zusammengefasst.

Eine Komponente

- ist ein Strukturierungsmittel und realisiert das Kapselungsprinzip.
- sollte eine hohe inhaltliche Kohäsion und eine schwache Kopplung zu ihrem Umfeld besitzen.
- ist eine in ihrem Anwendungsbereich unteilbare Verteilungseinheit.
- liegt in einem ausführbaren Zustand vor.
- ist als Black-Box nutzbar, ohne dass Wissen über ihre interne Realisierung benötigt wird.
- besitzt Importschnittstellen zum Zugriff auf benötigte Funktionalität und Exportschnittstellen zum Bereitstellen der von ihr erbrachten Funktionalität.
- besitzt nur explizite Abhängigkeiten zu ihrem Kontext.
- kann mit anderen Komponenten interagieren.

Die Anforderungen der hohen inhaltlichen Kohäsion und der schwachen Kopplung zum Umfeld finden sich implizit auch in anderen Eigenschaften wieder. Eine hohe inhaltliche Kohäsion bedeutet, dass die Komponente eine Abstraktion oder ein Konzept eines Anwendungsbereiches beschreibt. Auf dieser Ebene der Betrachtung ist sie unteilbar. Die schwache Kopplung erleichtert die Verteilung der Komponente und wird durch die Forderung nach ausschließlich expliziten Kontextabhängigkeiten begünstigt.

3.3 Services und Komponenten

Services oder Dienste sind im Bereich der Softwareentwicklung kein neues Konzept. Die Idee, Funktionalität in einem anderen Prozess, unter Umständen sogar auf einem entfernten Rechner aufrufen zu können, steht schon hinter dem RPC-Konzept (Remote Procedure Call, Entfernter Methodenaufruf[25]). Auch hier steht der funktionale Aspekt im Vordergrund und bildet die Basis für die Strukturierung auf einer hohen Abstraktionsebene.

Obwohl der konzeptuelle Ansatz also keineswegs neu ist, nehmen Services und **Serviceorientierte Architekturen** (SOA) seit einiger Zeit einen breiten Raum – gerade auch in den anwendungsorientierteren Veröffentlichungen – ein. Services sind zu einem Modewort geworden und vielfach zu einem Verkaufsargument der Marketingstrategen[26]. Der Grund für diese Wiederentdeckung liegt in einer technologischen Neuerung, den Webservices, begründet. Etwas verallgemeinernd dargestellt ermöglichen Webservices den Aufruf entfernter Dienste auf Basis von HTTP (Hypertext Transfer Protocol) und XML (eXtensible Markup Language). Diese technologische Basis ist auf vielen aktuellen Betriebssystem- respektive Hardwareplattformen verfügbar, folglich ist der Ansatz plattformübergreifend nutzbar und erlaubt die Interaktion von Softwareapplikationen über Betriebssystemgrenzen hinweg.

[25] Für weitere Informationen zu RPC siehe [CDK02].

[26] In [MPN+03] sind „Advanced Webservices“ knapp unterhalb des Höhepunktes im so genannten „Hype Cycle“ dargestellt. Der Höhepunkt wird als „Peak of Inflated Expectations“, als der Höhepunkt der überhöhten Erwartungen, bezeichnet.

Die Ideen hinter Services und Komponenten liegen dicht beieinander. Die genauere Betrachtung kann ein besseres Verständnis des Komponentenbegriffs und seiner vielfältigen konzeptionellen Facetten vermitteln und schließt sich daher in dieser Arbeit der Auseinandersetzung mit der Definition desselben an.

Der wesentliche Unterschied der beiden Konzepte liegt darin begründet, dass andere Aspekte in den Vordergrund gestellt werden. Services betonen eine funktionale Sichtweise. Diese Betrachtung liefert zwar auf einer hohen Abstraktionsebene ebenfalls eine Aufteilung eines Softwaresystems in kleinere Teile, aber anhand der Funktionalität. Komponenten betonen die statische Sicht beziehungsweise den Verteilungsaspekt. Die Zerlegung eines Systems in Komponenten liefert eine Aufteilung in Verteilungseinheiten, dies ist bei Services nicht der Fall.

Dieser letzte Punkt zeigt sich noch deutlicher, wenn man die Nutzung einer Komponente oder eines Services durch einen Client betrachtet. Während die Komponente zur Nutzung verteilt wird, nutzt der Client eines Services diesen an der Stelle, an der sich der Service befindet.

Die Betrachtung der Servicenutzung fördert weitere interessante Unterschiede zu Tage. Bei der Realisierung einer servicebasierten Architektur existiert in der Regel eine Vermittlungsinstanz. Ein Nutzer kontaktiert einen Service nicht direkt, sondern ermittelt über den Vermittler die zu kontaktierende Realisierung. Dieser Zwischenschritt erhöht die Flexibilität einer solchen Architektur beträchtlich und trägt zu einer losen Kopplung bei. Er erlaubt nicht nur einen für den Nutzer transparenten Wechsel – eine gleich bleibende Schnittstelle vorausgesetzt – der Realisierung eines Services, auch Lastverteilungsmechanismen können so in nahe liegender Weise umgesetzt werden. Für den Client sollte die konkrete Realisierung dabei nicht feststellbar sein. Nach Beendigung einer Transaktion kann der Client bei einem erneuten Aufruf desselben Dienstes an eine andere Realisierung verwiesen werden. Da dieses Verhalten gewünscht ist, darf der Wechsel keine negativen Folgen für die Ausführung der Funktionalität auf Seiten des Services oder für den aufrufenden Client haben.

Aufgrund dieser Betrachtung ergibt sich zwingend, dass die Realisierung eines Services keinen durch den Nutzer wahrnehmbaren Zustand über die Dauer einer Transaktion hinaus besitzen darf. Bei einem Wechsel der Realisierung wären anderenfalls kaum vorhersehbare Effekte möglich. In einigen Szenarien ist jedoch ein transaktionsübergreifender Zustand gewünscht, beispielsweise soll ein Nutzer beim wiederholten Aufruf eines Dienstes Informationen über frühere Nutzungen des Dienstes erhalten. Als Ausweg bietet sich hierbei die Nutzung eines externen Speichers für den Zustand – dies kann eine Datenbank sein – an. Dadurch wird es möglich, dass der Service – im Sinne der bereitgestellten Funktionalität, nicht einer konkreten Realisierung – einen Zustand besitzt. Dieser ist dann realisierungsübergreifend. Hier wird ein klarer Unterschied zu Komponenten deutlich, welche einen eigenen Zustand (vgl. Abschnitt 3.2.4, 40) besitzen können.

Der letzte Abschnitt macht deutlich, dass streng zwischen einem Service, also Funktionalität, welche durch eine wohldefinierte Schnittstelle bereitgestellt wird, und der Realisierung unterschieden werden muss. Eine Komponente kann einen oder mehrere Services bereitstellen, ein Service kann durch eine oder mehrere Komponenten realisiert sein[27]. Diese Komponenten werden – meist durch ihren Nutzer – instanziert, hier ist die Unterscheidung in Instanz und Typ sinnvoll. Bei der Betrachtung des Services ist sie es nicht. Folgendes Beispiel verdeutlicht diesen Punkt. Ein einfacher Service *CalculatorService* stellt Funktionalität zur Auswertung eines arithmetischen Ausdrucks bereit, dieser wird in Form eines Strings übergeben. Der Service liefert das Ergebnis der Auswertung. Die Funktionalität wird durch

[27] Es sind auch Services auf Basis anderer Programmierparadigmen als dem der Komponenten denkbar, CORBA ist ein gutes Beispiel für die Möglichkeit, Dienste auf Basis des prozeduralen Ansatzes zu realisieren. Um den Text nicht über Gebühr aufzublähen wird auf die wiederkehrende Nennung dieser Alternative verzichtet.

eine Komponente *CalculatorComponent* realisiert. Um den Dienst möglichst vielen Clients gleichzeitig zur Verfügung stellen zu können, wird diese mehrfach instanziert, die Instanzen werden in einem Pool zwischengespeichert und bei Bedarf werden zusätzliche erzeugt. In diesem Szenario existiert dennoch nur ein Dienst *CalculatorService*.

Ein Service ist definiert durch seinen Namen und eine Schnittstelle, welche seine Nutzung ermöglicht. Dies ist unabhängig von Realisierungen, deren Verteilung oder Instanzierung. Daher ist auch die Betrachtung der Identität eines Services im Sinne der Identität, wie sie in der Objektorientierung für Objekte definiert wird, nicht sinnvoll auf Services zu übertragen. Auch dies ist ein Unterschied zu Komponenten, wie sie in den vorhergehenden Abschnitten diskutiert wurden. Etwas vereinfachend lassen sich Services mit den aus der objektorientierten Programmierung bekannten Interfaces, Komponenten bzw. Komponententypen mit Klassen und Komponenteninstanzen mit Objekten vergleichen. Eine Komponente kann einen Service realisieren, eine Klasse ein Interface. Instanziert wird in einem Fall die Komponente, im anderen die Klasse um eine zur Laufzeit einer Anwendung nutzbare Entität zu erhalten.

Beim Aufruf von entfernten Diensten über ein Netzwerk ist es sinnvoll, die Anzahl der zur Nutzung eines Services nötigen Aufrufe möglichst klein zu halten. Dies muss sich in den Schnittstellen niederschlagen, hierbei sind möglichst große funktionale Einheiten in einem Aufruf zusammenzufassen.

Die Schnittstelle eines Dienstes muss publiziert sein, und ihre Beschreibung sollte getrennt von der Realisierung des Dienstes verfügbar sein, um eine Nutzung in der bisher schon dargelegten Art und Weise zu ermöglichen. Die Kenntnis dieser Schnittstelle muss für den Client ausreichend sein, um einen Dienst zu nutzen.

3.4 Fazit

Komponenten und auch Services erlauben die Strukturierung einer Softwareanwendung auf einer höheren Abstraktionsebene, als dies allein mit objektorientierten Konzepten der Fall ist. Gerade durch das Ausnutzen der losen Bindung eines Nutzers an eine Komponente oder einen Dienst lassen sich sehr interessante Anwendungen realisieren.

Die in diesem Kapitel vorgestellten Konzepte finden im weiteren Verlauf der vorliegenden Arbeit in zwei unterschiedlichen Bereichen Anwendung. Zum einen werden Komponenten und Services im Rahmen des entwickelten Prototyps eingesetzt, um eine flexible und erweiterbare Umgebung für prototypische Entwicklungen im Bereich der Endnutzerprogrammierung zu schaffen.

Zum anderen werden sie als Basis für ein konkretes Konzept zur Realisierung domänenspezifischer Sprachen verwendet, welches in dem angesprochenen Prototyp realisiert wird. In diesem Zusammenhang stellen Komponenten die Bausteine dar, aus denen ein Endnutzer eine Applikation entwickelt.

Der Begriff der Komponente wurde in diesem Kapitel, aufgrund seiner Bedeutung für die weitere Arbeit und da er in der Literatur nicht klar umrissen ist, ausführlich diskutiert um eine konzeptionelle Basis zu legen. Durch die Ergänzung um die vergleichende Betrachtung mit der Idee der Services ergibt sich eine grundlegende Klärung der Begrifflichkeiten, ohne die eine präzise Betrachtung kaum möglich ist.

Dieses Kapitel beschränkte sich dabei auf konzeptionelle, auf abstrakte Betrachtungen und ließ die Diskussion der unterschiedlichen Realisierungen bewusst aus. Hier ist, auch im Hinblick auf die eingangs gegebene Darstellung der möglichen Probleme bei der ausschließlichen Verwendung objektorientierter Ansätze, eine Ergänzung notwendig. Komponenten müssen nicht zwingend objekt-

orientiert entwickelt werden, in vielen der aktuellen Komponentenmodelle und auch in dem später in dieser Arbeit verwendeten ist dies jedoch der Fall. Durch ungeschickte Realisierungen können die für die Objektorientierung geltenden Einschränkungen auf das Komponentenmodell übertragen werden.

Ein Beispiel hierfür ist das angesprochene Problem der fragilen Oberklassen. Verwendet man das Prinzip der Vererbung zur Anpassung einer objektorientiert realisierten Komponente, so überträgt sich dieses Problem von den Klassen auf die Komponenten. Vererbungsbeziehungen zwischen Komponenten stellen eine sehr starke Form der Abhängigkeit dar und sind nicht notwendig aus den Schnittstellenbeschreibungen zu ersehen. Eine solche Kontextabhängigkeit ist daher – wenn sie nicht vermieden werden kann – gesondert in geeigneter Form zu dokumentieren.

Dieses Kapitel repräsentiert das Ende des ersten Abschnittes der vorliegenden Arbeit. Der nun folgende zweite Abschnitt behandelt den Entwurf und die Realisierung der oben angesprochenen Endnutzerprogrammierumgebung. Darüber hinaus wird die getroffene Auswahl einer Implementationssprache und deren Eignung zur Realisierung und Nutzung von Komponenten im Sinne der Definition dieses Kapitels dargelegt.

4 Entwurf einer flexiblen Endnutzerprogrammierumgebung

„Es ist nichts beständig als die Unbeständigkeit.“
Immanuel Kant, deutscher Philosoph, 1724-1804

Dieses Kapitel beschreibt den Entwurf eines Anwendungsframeworks für Endnutzerprogrammierumgebungen, dieses wird im Folgenden auch als das Cobamos-Framework bezeichnet. Zunächst wird dieses Vorhaben kurz motiviert und es werden die notwendigen Anforderungen an eine solche Entwicklung erhoben.

Hieran schließt sich eine Beschreibung des Entwurfs an. Wie Kant schon – ohne die Anwendbarkeit seiner Aussage in der Softwareentwicklung zu ahnen – schrieb, ist die Unbeständigkeit ein ständiger Faktor. In Softwareprojekten sind sich ändernde Anforderungen und daraus resultierende Notwendigkeiten ein wichtiger Aspekt. Dies wird im Entwurf des Frameworks durch Konzepte zur Bereitstellung von wieder verwendbarer Funktionalität durch das Framework und Möglichkeiten zur Anpassung und Erweiterung der vorhandenen Funktionalität berücksichtigt. Hier ist zum einen der Einsatz einer serviceorientierten Architektur innerhalb der Anwendung und zum anderen eine komponentenorientierte Variante des Kommando-Musters zu nennen.

Den Abschluss des Kapitels bildet die Vorstellung des in dieser Arbeit erarbeiteten Konzeptes zur Kapselung von Endnutzerprogrammiermodellen in Komponenten. Dieser Ansatz ermöglicht es, unterschiedliche Modelle als Grundlage der entworfenen Entwicklungsumgebung zu verwenden.

4.1 Einleitung

Im Rahmen der vorliegenden Arbeit soll ein Ansatz auf Komponentenbasis erarbeitet werden, der es Endnutzern ermöglicht, in begrenztem Rahmen Softwareanwendungen zu entwickeln. Damit ein Endnutzer eine Softwareapplikation entwickeln kann, ist ein entsprechendes Entwicklungswerkzeug vonnöten.

Soll nur eine einzelne konkrete Anwendung durch den Endnutzer anpassbar sein, so reicht es aus, in diese entsprechende Möglichkeiten zu integrieren. Dies kann – je nach Kenntnisstand der Zielgruppe – beispielsweise durch eine in das Produkt integrierte Skriptsprache oder durch dialoggestützte Anpassungsmechanismen geschehen. Sollen jedoch eigenständige Anwendungen entwickelt werden, so ist es sinnvoll, eine autonome Entwicklungsumgebung zur Verfügung zu stellen. Diese sollte sowohl den Entwurf einer Applikation als auch das Überführen eines Modells in eine ausführbare Form ermöglichen, um den Nutzer von der Notwendigkeit zu befreien, zum Erreichen seiner Ziele mit mehreren unterschiedlichen Werkzeugen interagieren zu müssen. In der Informatik hat sich der Begriff **Integrierte Entwicklungsumgebung** (IDE, Integrated Development Environment) für solche Werkzeuge etabliert. Während die ersten IDEs lediglich die Integration von Texteditor und Compiler bzw. Linker in einer Anwendung boten, so unterstützen aktuelle IDEs den Softwareentwickler in der Regel auch bei darüber hinausgehenden Aufgaben, wie beispielsweise dem Entwurf einer Nutzungsoberfläche.

Im Rahmen vorhergehender Projekte am Institut für Informatik der Johannes Gutenberg-Universität Mainz wurden bereits einige Modellierungsumgebungen im Bereich der Sportinformatik realisiert. In diesen Projekten wurden – auch aufgrund unterschiedlicher Ansätze – jeweils neue Anwendungen erstellt, um die erarbeiteten Konzepte prototypisch umzusetzen. Erfahrungsgemäß sind sich große Teile solcher Anwendungen jedoch ähnlich. So werden beispielsweise immer Funktionalitäten zur Verwaltung von Konfigurationseinstellungen, zur Kommunikation mit dem Nutzer der Applikation

oder zur Bereitstellung von Protokolldateien benötigt. Die Notwendigkeit, diese Bestandteile einer Anwendung wiederholt zu realisieren, steht einer Konzentration auf die eigentlichen Projektziele entgegen. Hier könnte eine geeignete Wiederverwendung von Anwendungsteilen eine Entlastung der Softwareentwickler von Routineaufgaben bewirken. Dadurch stünde mehr Zeit für die Realisierung der eigentlichen Ansätze zur Verfügung.

Diese Erfahrungen und die Absicht, im Rahmen der vorliegenden Arbeit Untersuchungen zu den Möglichkeiten im Bereich der komponentenbasierten Endnutzerprogrammierung anzustellen, motivieren den Entwurf einer anpassbaren und flexiblen Entwicklungsumgebung. Diese soll zum einen der prototypischen Umsetzung der Konzepte dieser Arbeit dienen, zum anderen aber auch als Basis für zukünftige Projekte in diesem Feld nutzbar sein.

Um eine solche Nutzung zu ermöglichen, wird im Folgenden keine Softwareanwendung entworfen. Vielmehr wird ein Anwendungsrahmen, welcher erst durch Erweiterungen und Anpassungen zu einer konkreten Anwendung wird, entwickelt. Für solche Gerüste hat sich der Begriff Framework (engl. Rahmen, Rahmenwerk) etabliert (vgl. z. B. [Kai05]). Der Begriff wird zwar in der Literatur unterschiedlich verwendet, zum Teil werden auch Klassenbibliotheken so bezeichnet (vgl. [Bra04]), im Verlauf dieser Arbeit bezeichnet ein Framework jedoch immer einen Anwendungsrahmen oder eine generische Anwendung.

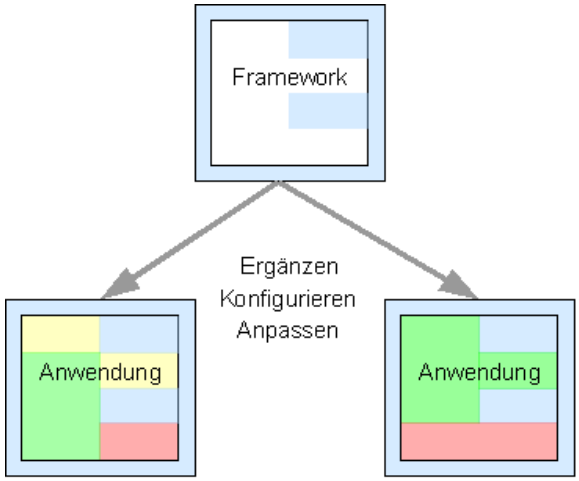


Abb. 4.1: Der Zusammenhang zwischen einem Framework und den darauf basierenden Anwendungen

Der Zusammenhang zwischen einem Framework und den darauf basierenden Anwendungen ist in Abb. 4.1 dargestellt. Das Framework stellt einen Anwendungsrahmen bereit und definiert Erweiterungs- und Anpassungsmechanismen. Da das Framework unterschiedliche Anpassungen ermöglicht, ist es die Basis für eine Familie von Anwendungen[28]. Im Unterschied zu der Verwendung einer Klassenbibliothek, deren Funktionalität durch die Anwendung aufgerufen wird, werden bei der Verwendung eines Frameworks die anwendungsspezifischen Teile von diesem aufgerufen. Dies geschieht beispielsweise dadurch, dass das Framework eine zu implementierende Schnittstelle vorgibt. Zur Verwendung des Frameworks wird diese Schnittstelle durch eine anwendungsspezifische Realisierung implementiert. Bei der Ausführung der resultierenden Anwendung wird diese Realisierung dann vom Framework aufgerufen und die angepasste Funktionalität ausgeführt. Die

[28] Der Begriff der Anwendungsfamilie wird in einem Exkurs zur vorliegenden Arbeit in Appendix A diskutiert.

Aufrufe erfolgen dabei vom wieder verwendeten Teil der Applikation hin zu den neu implementierten. Dieser Umstand wird als **inversion of control** (engl. Umkehrung der Steuerung) bezeichnet[29] [FS97].

Da nicht absehbar ist, in welchen Zusammenhängen bzw. Projekten das hier entworfene Framework für Endnutzerprogrammierumgebungen Anwendung finden wird, sollten möglichst geringe Vorgaben hinsichtlich des zu verwendenden Programmiermodells oder seiner Realisierung gemacht werden. Eine Anpassung sollte möglich sein, ohne dass bestehende Teile der Anwendung im Quelltext angepasst werden müssen. Idealerweise sollte auch eine erneute Übersetzung aus anderen Gründen (z. B. Aktualisieren von Verweisinformationen) unnötig sein.

In der unten stehenden Abbildung 4.2 wird das in diesem Kapitel entworfene Framework (ausgewählte Details zur Realisierung des Entwurfs werden in Kapitel 6 erläutert) in den Gesamtkontext der Resultate dieser Arbeit und der intendierten Nutzergruppe eingeordnet. Das Cobamos-Framework ist ausschließlich zur Nutzung durch Softwareentwickler gedacht, welche auf dieser Basis Applikationen entwickeln. Der Endanwender einer solchen Applikation muss jedoch beim Entwurf des Frameworks ebenfalls berücksichtigt werden, da Entwurfsentscheidungen an dieser Stelle die Nutzbarkeit der resultierenden Anwendungen beeinflussen können. An den entsprechenden Stellen im Text wird die Unterscheidung – wenn sie sich nicht aus dem Zusammenhang erschließt – zwischen dem Entwickler als Nutzer des Frameworks und dem Endanwender als Nutzer einer auf dem Framework basierenden Applikation kenntlich gemacht.

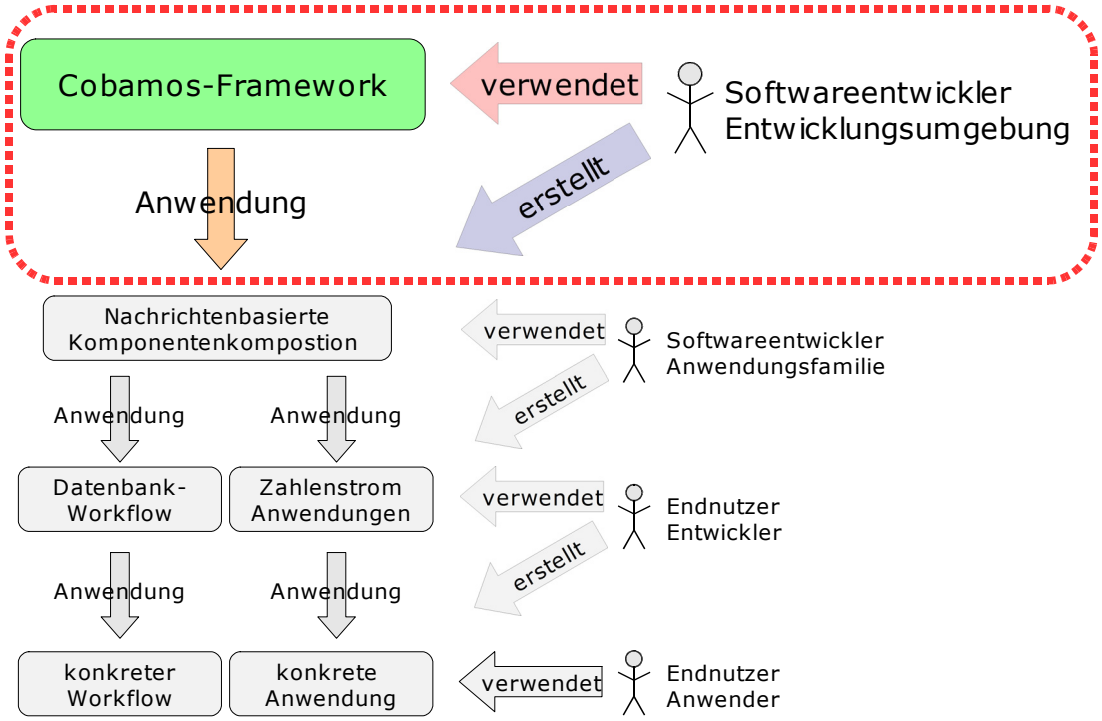


Abb. 4.2: Einordnung der in Kapitel 4 dargestellten Ergebnisse in den Gesamtkontext

Im weiteren Verlauf der Arbeit wird die Realisierung einer Programmierumgebung für Endnutzer als eine Anwendung der Ergebnisse rund um das Cobamos-Framework gezeigt. Dies motiviert sich aus der Aufgabenstellung. In der Abbildung entspricht diese Anwendung des Frameworks der zweiten Ebene von oben.

[29] Etwas scherzhaft auch als „Hollywood-Prinzip“ bezeichnet, „don’t call us, we’ll call you“.

Der folgende Abschnitt präzisiert nun die in dieser Einführung teilweise bereits angeklungenen Anforderungen, um sie als Ausgangspunkt für den Entwurf nutzen zu können.

4.2 Anforderungen an das Cobamos-Framework

Auf Anforderungen, welche generell an das Design eines Frameworks gestellt werden sollten, wird an dieser Stelle nicht eingegangen. Ein sehr gutes und umfassendes Werk zu diesem Thema ist [CA06]. Obwohl die dort vorgestellten Konventionen und Muster auf Microsofts .NET-Framework basieren, werden die allgemeinen Prinzipien ausreichend behandelt und mit Beispielen aus der Realisierung des .NET-Frameworks ergänzt.

4.2.1 Unterstützung unterschiedlicher Programmiermodelle

Bei der Endnutzerprogrammierung geht es, wie bei der Programmierung allgemein, darum, ein Modell der Realität zu schaffen. Es sind unterschiedliche Ausprägungen von Modellen denkbar, auf deren Basis sich eine Anwendung realisieren lässt. Hier sollten möglichst geringe Vorgaben bzw. Einschränkungen durch das Framework vorgegeben werden.

4.2.2 Wiederverwendung

Da die im Rahmen dieser Arbeit entwickelte Anwendung auch für weitere Entwicklungen auf dem Gebiet der Endnutzerprogrammierung als Basis nutzbar sein soll, ist die Wiederverwendung möglichst umfangreicher Teile der Funktionalität der Entwicklungsumgebung ein wichtiges Ziel. Dies muss aber in einer Form geschehen, die möglichst wenige Einschränkungen für die zukünftigen Ansätze bedingt.

4.2.3 Anpassbarkeit

Eine Programmierumgebung für Endnutzer sollte auf die speziellen Bedürfnisse der Zielgruppe ausgerichtet sein. Da es sich bei Endnutzern nicht um eine homogene Gruppe handelt (vgl. Klassifikation der Endnutzer in Kapitel 2), kann ein solches Entwicklungswerkzeug entweder nur auf eine beschränkte Gruppe ausgerichtet sein, oder es muss für unterschiedliche Gruppen konfigurierbar sein.

Neben der Anpassung an eine Zielgruppe sollte auch eine Anpassung an eine neue Herangehensweise oder ein anderes Programmiermodell für die Endnutzerprogrammierung möglich sein.

Zudem kann der Fall eintreten, dass die durch das Framework bereitgestellte Funktionalität angepasst oder ergänzt werden muss. Die verwendete Architektur sollte also auch an dieser Stelle Änderungen erlauben.

Während sich die generelle Forderung nach Anpassbarkeit auf Anpassungen durch Softwareentwickler bezieht, ist hier auch die Erweiterung von Funktionalität einer auf dem Framework basierenden Anwendung durch einen Endanwender zu beachten. Für eine Verwendung einer Programmierumgebung im Bereich der Endnutzerprogrammierung wäre es wünschenswert, dass Anpassungen der Funktionalität teilweise durch den Anwender möglich sind. Eine Anpassung durch einen Endanwender darf aber beispielsweise keine Änderungen am Quelltext der Anwendung erfordern. Die Möglichkeiten zur Anpassungen müssen so gestaltet sein, dass sie ohne Kenntnis der Anwendungsrealisierung vorgenommen werden können und sollten möglichst wenig Interaktion seitens des Endanwenders erfordern.

4.2.4 Erweiterbarkeit

Neben der Anpassung von vorhandener Funktionalität soll es das Framework auch erlauben, neue Funktionalität hinzuzufügen. Idealerweise ist dies auch nach dem so genannten Auslieferungszeitpunkt möglich.

Es ist zu erwarten, dass diese Anforderungen nur unter Einschränkungen – beispielsweise der Vorgabe von zu erfüllenden Schnittstellen – erfüllt werden kann und kein „beliebiges“ Hinzufügen neuer Funktionalität möglich ist.

4.2.5 Einfache Nutzung

Unter dem Gesichtspunkt der einfachen Nutzung sind zwei Aspekte zu unterscheiden. Zum einen ist die einfache Nutzbarkeit des Frameworks durch einen Entwickler zu berücksichtigen. Dazu muss die spätere Verwendung ausreichend beim Entwurf einbezogen werden. Dies macht beispielsweise eine entsprechende Gestaltung der Schnittstellen notwendig. In diesem Zusammenhang ist [CA06] zu erwähnen, hier wird über Schwierigkeiten von Entwicklern im Umgang mit bestimmten Bereichen der ersten Version des .NET-Frameworks der Firma Microsoft (dieses wird in Kapitel 5 besprochen) aufgrund der Schnittstellengestaltung berichtet.

Daneben soll die einfache Nutzung der später mit diesem Framework realisierten Anwendung durch einen Endnutzer möglich sein. Dies kann durch den Frameworkentwurf unterstützt werden. Prinzipiell ist aber immer eine Verwendung des Frameworks in einer Form denkbar, die zu einer übertrieben komplizierten Endanwendung führt. Diese Möglichkeit kann durch den Entwurf nicht unterbunden werden.

4.3 Entwurf des Frameworks

4.3.1 Allgemeines

Dieser Abschnitt stellt die in der Entwicklungsumgebung umgesetzten Ansätze zur Erfüllung der Anforderungen aus Abschnitt 4.2 dar. Die Ausführungen konzentrieren sich dabei auf die konzeptionell interessanten Stellen des Entwurfs, eine vollständige und detaillierte Darstellung des Entwurfs würde an dieser Stelle über den Rahmen hinausgehen. Es wird bewusst noch keine konkrete Realisierung eines Konzeptes zur Endnutzerprogrammierung verfolgt. Der entsprechende Teil der IDE, im Text als Programmiermodell bezeichnet, wird zunächst als Black-Box betrachtet. Ein Konzept zur Umsetzung einer derartigen Kapselung für ein Programmiermodell wird in den anschließenden Abschnitten vorgestellt. Es wird zu diesem Zeitpunkt soweit als möglich vermieden, Annahmen über die Interna der Realisierung zu machen. Wenn die hier vorgestellten Ansätze Einschränkungen bzw. Anforderungen an das Programmiermodell bedingen, wird explizit darauf hingewiesen.

Da vermieden werden soll, dass der entwickelte Ansatz zu sehr an einer bestimmten Form der Implementierung oder an einer programmiersprachenspezifischen Umsetzung entlang verläuft, wird in diesem Kapitel nur eine Architektur für das Gesamtframework und bestimmte Teilsysteme desselben entworfen. Unter dem Begriff der Architektur einer Anwendung wird im Rahmen dieser Arbeit die Beschreibung der Aufteilung einer Applikation in Teilsysteme und Komponenten, die Festlegung der Aufgaben dieser, die Beschreibung der Interaktionen und die Spezifikation von benötigten Schnittstellen verstanden. Diese Betrachtungen erfolgen auf einer Abstraktionsebene, welche es erlaubt die vorgestellten Überlegungen auf andere Implementationen zu übertragen. Auf die

prototypische Implementation, die im Rahmen dieser Arbeit vorgenommen wurde, wird in Kapitel 6 eingegangen, hier finden sich dann auch Einzelheiten zur möglichen Realisierung.

Dieser Entwurf verwendet in großem Maße das in Kapitel 3 besprochene Komponentenkonzept und die Idee der Serviceorientierung. Das Framework wird derart entworfen, dass die anwendungsspezifische Anpassung durch den Einsatz von Komponenten vorgenommen wird. Da Komponenten Verteilungseinheiten darstellen, eignen sie sich gut, um die anwendungsspezifischen Teile zu kapseln. Durch die expliziten Schnittstellen und die schwache Kopplung zum Umfeld, die Teil des Komponentenkonzeptes sind, wird eine leichte Austauschbarkeit dieser Anwendungsbestandteile erreicht. Dies bedingt in der Folge eine bessere Anpassbarkeit der Anwendung.

In Kapitel 3 wurde auf Seite 44 bereits kurz angesprochen, dass bei einer objektorientierten Realisierung von Komponenten darauf zu achten ist, die in rein objektorientierten Entwürfen potenziell vorhandenen Probleme nicht zu übertragen. Dies betrifft beispielsweise die Abhängigkeiten von der Umgebung; eine Komponente sollte eindeutig definierte Schnittstellen besitzen und nur explizite Abhängigkeiten von ihrem Kontext aufweisen. Im Fall einer aus mehreren Klassen bestehenden Komponente ist es denkbar, dass die von einem Nutzer anzusprechende Funktionalität auf mehrere Klassen verteilt ist und dadurch unnötige Abhängigkeiten der Komponente zum Umfeld entstehen. Durch Verwendung des so genannten Fassadenmusters (siehe [Gam95]), eines Strukturmusters, ist es leicht möglich, dieses Problem zu beheben. Da in dieser Arbeit eine Umsetzung des Entwurfs mit objektorientierten Mitteln verfolgt wird, ist dieses Muster hilfreich und findet an einigen Stellen Anwendung. Auch bei der Realisierung von Komponenten zur Anpassung des IDE-Frameworks sollte wenn nötig auf dieses Muster zurückgegriffen werden.

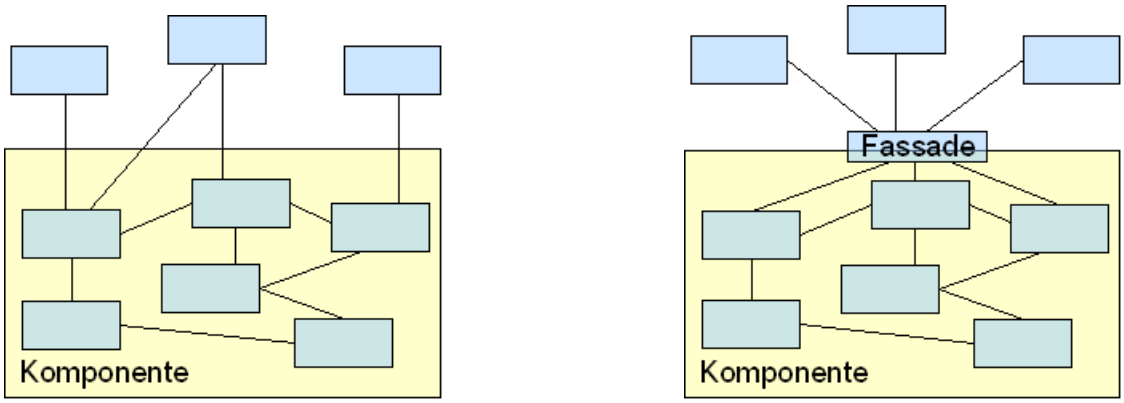


Abb. 4.3: Das Strukturmuster „Fassade“

Die Abb. 4.3 verdeutlicht das Fassadenmuster. Die linke Seite der Grafik zeigt den Zustand vor der Anwendung des Musters. Die Linien symbolisieren Interaktionen, die Rechtecke Klassen. Die außerhalb der Komponente befindlichen Nutzer kommunizieren mit mehreren komponenteninternen Klassen, die entstehenden Abhängigkeiten schränken die Möglichkeiten zur Weiterentwicklung der Komponente ein und erschweren die Übersicht über die existierenden Abhängigkeiten. Auf der rechten Seite wird eine Fassadenklasse verwendet, welche die Aufgabe hat, Aufrufe an die entsprechenden Klassen innerhalb der Komponente weiterzuleiten. In der Folge können Änderungen an den internen Klassen der Komponente vorgenommen werden, ohne dass Anpassungen an den Nutzern notwendig werden. Änderungen an den Schnittstellen der komponenteninternen Klassen erfordern lediglich eine Anpassung in der Fassadenklasse, die Schnittstelle der Komponente kann beibehalten werden. Dieses Vorgehen ist vergleichbar mit den durch das Geheimnisprinzip (z. B. [Mey97]) nahe gelegten

Entwurfsrichtlinien in der objektorientierten Programmierung. Durch das Fassadenmuster wird dieses Prinzip auf die Ebene der Komponenten gehoben, die Fassadenklasse – oder genauer ihre Schnittstelle – realisiert dabei die Schnittstelle der Komponente.

In der Objektorientierung werden Wiederverwendung und Anpassung häufig durch Vererbung realisiert. Dies ist auch in objektorientierten Frameworks der Fall, hier werden Basisklassen bereitgestellt, welche von den Nutzern des Frameworks zur Anpassung erweitert werden. Eine solche Vorgehensweise hat den Vorteil, dass den späteren Nutzern des Frameworks möglichst viel Arbeit durch das Bereitstellen von Standardimplementierungen abgenommen wird. Insbesondere im Bereich der grafischen Nutzungsoberflächen ist dieser Ansatz verbreitet (siehe zum Beispiel [GA03]). Er birgt jedoch die Gefahr, dass die Unterklasse durch das Überschreiben von Eigenschaften der Oberklasse Frameworkfunktionalität zerbricht.

Aber auch im Falle einer abstrakten Basisklasse kann die Verwendung von Vererbung als Mechanismus zur Anpassung in einem komponentenbasierten Framework zu Problemen führen, da dies eine sehr hohe Kopplung der spezialisierten Klassen – und damit der Komponente – an das Framework bzw. die Basisklasse bedingt. Darüber hinaus führt eine Vererbungsbeziehung über die Grenzen der Komponente hinweg zur Übertragung des Problems der fragilen Oberklasse auf die Ebene der Komponenten. Dies sollte vermieden werden.

Der umgesetzte Ansatz zur komponentenbasierten Anpassung des Frameworks vermeidet daher so weit als möglich diese in objektorientierten Frameworks üblicherweise verwendeten Mechanismen zur Anpassung bzw. zur Realisierung von anwendungsspezifischem Verhalten. Um eine möglichst lose Kopplung der Komponenten und des Frameworks zu erhalten, werden die benötigten Interaktionsschnittstellen in Form von Interfaces[30] beschrieben. Obwohl man die Implementation von Interfaces durchaus als eine besondere Form der Vererbung ansehen kann, so entfallen doch die oben angesprochenen Probleme bzw. werden auf die Notwendigkeit einer feststehenden Schnittstelle reduziert, da es sich nicht um eine Vererbungsbeziehung im Sinne der Weitergabe von Implementation handelt. Hier ist anzumerken, dass eine Arbeitsentlastung der Softwareentwickler bei der Anwendung des Frameworks dadurch erreicht werden kann, dass zu den Schnittstellen Standardimplementationen bereitgestellt werden, welche aber außerhalb des eigentlichen Framework angesiedelt sind und diesem nicht bekannt sind, daher auch nicht von diesem genutzt werden. Dadurch wird das Problem vermieden, dass durch die Implementation einer Unterklasse Funktionalität des Frameworks zerbrechen kann. Da das Problem der fragilen Oberklasse jedoch weiterhin besteht, ist hier abzuwägen, ob dies in einem überschaubaren Entwicklungsrahmen vertretbar ist. In den folgenden Entwurfsüberlegungen werden stellenweise solche Klassen bereitgestellt.

Die Interfaces werden als eigenständige Komponenten realisiert, welche sowohl vom Framework als auch von den anwendungsspezifischen Komponenten genutzt werden können. Als weiterer Vorteil ergibt sich hieraus die Möglichkeit, die anwendungsspezifischen Komponenten ohne Kenntnis des Frameworks entwickeln und übersetzen zu können.

[30] In einigen objektorientierten – insbesondere in aktuelleren – Frameworks findet dieser Ansatz ebenfalls Anwendung. Als Beispiel sei das Enterprise Java Beans Framework (EJB) [Sun06] von Sun angeführt. Bei diesem Framework handelt es sich um eine objektorientierte Realisierung für ein Komponentenframework.

4.3.2 Dynamisches Laden von Komponenten und Reflexion

Bei den Ausführungen zu Komponenten in Kapitel 3 wurde eine realisierungsneutrale Sichtweise verfolgt und die Aufmerksamkeit auf die konzeptionellen Wesenszüge des Begriffs der Komponente gelegt. Für den im Folgenden beschriebenen Entwurf werden Eigenschaften eines Komponentenmodells verwendet, welche aus diesem Grunde nicht im Rahmen der Begriffsdefinition angesprochen wurden. Es handelt sich dabei um die Möglichkeiten zur Reflexion und zum dynamischen Laden von Komponenten (zur Realisierung siehe Abschnitt 6.2). Da eine detaillierte Behandlung der beiden Konzepte an dieser Stelle über den Rahmen des Nötigen hinausgehen würde, wird hier nur versucht, die jeweiligen Ideen aufzuzeigen. Beide sind nicht charakteristisch für ein bestimmtes Komponentenmodell, ihre Verwendung stellt also keine Einschränkung auf eine einzige Realisierungsmöglichkeit dar. Darüber hinaus ließen sich die später angestellten Überlegungen auch bei einem Verzicht auf diese Möglichkeiten anpassen, die Folge wäre lediglich der Verlust einer gewissen Eleganz.

Reflexion bezeichnet die Möglichkeit in einem Programm Metainformationen über das Programm selbst bzw. die Entitäten (Klassen, Methoden,…), aus denen es aufgebaut ist, ermitteln zu können. Im Zusammenhang mit Komponenten ist diese Möglichkeit von Interesse, weil so eventuell zur Nutzung der Komponente erforderliche Metadaten nicht explizit durch geeignete Beschreibungen bereit gestellt werden müssen. Sie können stattdessen aus der Komponente durch Reflexion gewonnen werden. Hierzu gehören beispielsweise unterstützte Schnittstellen und vorhandene Methoden. An dieser Stelle ist anzumerken, dass Szenarien denkbar sind, in denen zusätzliche, nicht durch Reflexion zu gewinnende Informationen über eine Komponente zu deren Nutzung erforderlich sind. Solche Metadaten müssen dann in geeigneter Form – beispielsweise durch zusätzliche externe Beschreibungen – abgelegt werden. In diesen Bereich fallen beispielsweise Informationen zum Laufzeitverhalten einer Komponente und zu anderen Qualitätsmerkmalen. Neben diesen Fällen, in denen die Reflexion nicht hinreichend ist, weil sie zu wenig Informationen liefert, ist es auch möglich, dass zu viele Informationen durch den Reflexionsmechanismus ermittelt werden. Wird zum Beispiel das Fassadenmuster (siehe Abb. 4.3) zur Realisierung der Schnittstelle einer Komponente verwendet, so soll der Zugriff nur über die eigens eingeführte Fassadenklasse erfolgen. Falls der Reflexionsmechanismus – und dies ist in den gängigen Programmiersprachen der Fall – auf Ebene der Klassen realisiert ist, so können Informationen über Klassen ermittelt werden, welche nicht zur Schnittstelle gehören. Je nach Realisierung[31] der in der Komponente enthaltenen Klassen ist damit aber auch ein Zugriff auf die Klassen bzw. ihre Methoden und Attribute möglich. Dies ist jedoch die Basis für eine potenzielle Verletzung des Geheimnisprinzips und daher problematisch.

Viele der aktuellen Programmiersprachen bieten Möglichkeiten zur Reflexion, Java und C# sind bekannte Vertreter dieser Gruppe. In diesen Sprachen ist es möglich, zur Laufzeit Typinformationen über Objekte abzufragen. Diese Möglichkeiten übertragen sich auf komponentenbasierte Anwendungen, die mit diesen Sprachen realisiert werden. Über die reine Abfrage von Metainformationen bezüglich Typen, Methoden, Attributen und Ähnlichem hinaus stellen sowohl C# als auch Java Sprachkonstrukte bereit, um zusätzlich selbstdefinierte Metainformationen in den Programmtext

[31] Abhängig von der zur Realisierung der Komponente gewählten Programmiersprache kann es möglich sein, die Zugriffsrechte der Methoden bzw. Attribute einer Klasse so einzuschränken, dass ein Zugriff nur aus Klassen innerhalb derselben Komponente erlaubt ist. Diese Möglichkeit existiert beispielsweise in der Programmiersprache C# für Assemblys (vgl. Abschnitt 5.3). Dadurch reduziert sich das angesprochene Problem auf die reine Kenntnis unnötiger Details, ohne dass eine Gefahr besteht, dass das Geheimnisprinzip verletzt werden könnte.

einzubetten. In C# kann dies durch so genannte Attribute erreicht werden, Java kennt seit der Version 1.5 für diesen Zweck Annotationen.

Die Fähigkeit, eine zur Entwurfszeit unbekannte Komponente erst zur Laufzeit der Anwendung hinzuzufügen, wird als dynamisches Laden bezeichnet. Diese Möglichkeit geht über das aus objektorientierten Sprachen bekannte dynamische Binden einer konkreten Methodenrealisierung zur Laufzeit der Anwendung hinaus, da hierbei die Realisierung zur Übersetzungszeit im Programm bereits vorhanden ist.

Um die Nutzung einer zur Entwurfszeit unbekannten Komponente zu ermöglichen, muss entweder die zu verwendende Schnittstelle unabhängig von der Komponente festgelegt werden und zur Entwurfszeit bekannt sein, oder es müssen Mechanismen zum Entdecken der Schnittstelle zur Laufzeit vorhanden sein. Dies ist beispielsweise durch die oben angesprochene Möglichkeit der Reflexion gegeben. Bei einem solchen Ansatz kann kein Binden an eine Schnittstelle zur Entwurfszeit erfolgen und die Aufrufsyntax der Methoden der Schnittstelle ist unbekannt. Neben dem Laden der Komponente muss auch der Aufruf von zur Entwurfszeit unbekannten Methoden möglich sein.

Die gängigen Komponentenmodelle unterstützen das dynamische Laden in unterschiedlichem Maße. Microsofts Component Object Model (COM) ermöglicht beispielsweise das Laden einer unbekannten Komponente, dabei kann über einen als Automation bezeichneten Mechanismus auch eine Interaktion über zum Entwurfszeitpunkt unbekannte Schnittstellen erfolgen [GT00]. In der Programmiersprache Java steht durch die abstrakte Klasse `java.lang.ClassLoader` die Möglichkeit zum dynamischen Laden von Klassen [Sun04] zur Verfügung. Diese Möglichkeit besteht in der Folge auch in Komponentenmodellen auf der Basis von Java. Die Sprache C# mit ihrem Assemblykonzept (siehe Abschnitt 5.3) erlaubt ebenfalls ein dynamisches Laden von Assemblys und den darin enthaltenen Typen zur Laufzeit einer Anwendung.

4.3.3 Serviceorientierte Anwendungsarchitektur

Objektorientierte Frameworks basieren – wie eingangs erwähnt – für gewöhnlich auf einem als **inversion of control** bezeichneten Kommunikationsmuster. Hieraus resultiert im angedachten Einsatzszenario des Frameworks jedoch ein Problem. Wenn die Aufrufe von Funktionalität nur vom IDE-Framework hin zu den anwendungsspezifischen Teilen der Anwendung gerichtet sein können und Vererbungsbeziehungen von den Komponenten zum Framework vermieden werden sollen, so kann die bereitgestellte Basisfunktionalität von letzteren nicht genutzt werden. Dies ist aber eine der Anforderungen, daher kann keine reine Umsetzung eines Frameworkansatzes zum Einsatz kommen.

Eine Möglichkeit wäre es, neben dem Framework eine Klassenbibliothek mit der bereitzustellenden Funktionalität zu entwickeln. Sowohl das Framework als auch die anwendungsspezifischen Komponenten könnten dann unter Nutzung dieser Klassenbibliothek entwickelt werden. Diese Form der Wiederverwendung soll hier jedoch nicht weiter betrachtet werden. Um allen Komponenten der Entwicklungsumgebung gemeinsam nutzbare Funktionalität bereitzustellen, ohne dabei eine Bindung an eine konkrete Realisierung nötig zu machen, wird eine serviceorientierte Architektur innerhalb des Anwendungsrahmens umgesetzt. Durch die Verwendung des serviceorientierten Ansatzes wird eine lose Kopplung zwischen den beteiligten Entitäten erreicht.

Die Bereitstellung der Basisfunktionalität ist auch auf anderem Wege denkbar, als Alternative ist beispielsweise die Einführung so genannter **Kontextobjekte**[32] zu erwähnen. Dabei wird einer Komponenteninstanz, beispielsweise mittels des Aufrufs einer speziellen Methode, seitens des

[32] Ein vergleichbares Vorgehensmuster wird bei [Fow04] als **Dependency Injection** bezeichnet.

Frameworks ein Kontextobjekt übergeben. In der Regel ist zu einem solchen Objekt nur die Schnittstelle bekannt, so dass auch hier eine Bindung an eine konkrete Implementation vermieden wird. Dieses Kontextobjekt ermöglicht der Komponente – quasi als Vermittler – Zugriff auf das Framework bzw. Teile seiner Funktionalität. Die Verwendung von Kontextobjekten und einer serviceorientierten Architektur schließen sich gegenseitig keineswegs aus, so ließe sich der weiter unten angesprochene Vermittler für die Dienste auch durch ein geeignetes Kontextobjekt realisieren.

Die im Normalfall für verteilte Systeme verwendete serviceorientierte Architektur wird im hier vorgestellten Entwurf auf eine lokale Anwendung übertragen. Beispiele für den Einsatz service-orientierter Architekturen innerhalb lokaler Anwendungen finden sich sowohl im .NET-Framework als auch in der Entwicklungsumgebung SharpDevelop (siehe [HKS04]). Im .NET-Framework ist diese Architektur leider nicht sehr detailliert dokumentiert, einen Einstieg bietet die Dokumentation der relevanten Schnittstellen [Mic06a] [Mic06b].

Eine servicebasierte Architektur soll die Nutzung von Funktionalität erlauben, ohne dass zum Entwurfszeitpunkt bekannt sein muss durch welche Realisierung diese erbracht wird. Die Realisierungen der erbrachten Funktionalität werden als Services (oder Dienste) bezeichnet. Um die Unabhängigkeit von einer konkreten Realisierung zu ermöglichen, spricht ein Nutzer einen Dienst nicht direkt an, sondern erfragt zunächst über eine Vermittlerinstanz (diese wird auch als Registry bezeichnet) eine Realisierung für den Service (vgl. Abschnitt 3.3). Falls die Realisierung eines Services ausgetauscht wird, ist diese Änderung für den Nutzer transparent, da er beim nächsten Aufruf von der Vermittlerinstanz an die neue Realisierung verwiesen wird. Dieses Prinzip wird auch im Cobamos-Framework umgesetzt.

Die einfachste Realisierung einer Vermittlerinstanz stellt für jeden von ihr bereitgestellten Dienst eine Methode bereit, welche bei ihrem Aufruf eine Referenz auf eine Realisierung desselben zurückliefert. Falls neue Dienste in das Angebot aufgenommen werden müssen, so ist eine Anpassung der Implementation der Registry notwendig. Daher eignet sich dieses Vorgehen eher für Systeme, in denen Erweiterungen nicht zu erwarten sind.

Alternativ kann die Abfrage des Nutzers über ein geeignetes Identifikationsmerkmal für den Service erfolgen. Dieses Merkmal muss jedem potenziellen Nutzer bekannt sein, anderenfalls kann er den Service nicht nutzen. Da dies auch für die Schnittstellenbeschreibung des Dienstes gilt, und diese innerhalb des Frameworks auf jeden Fall eindeutig sein muss, wird sie als Identifikationsmerkmal verwandt. Die Vermittlerinstanz stellt eine Methode der Form `GetService(ServiceSchnittstelle)` bereit und liefert eine Referenz auf eine Realisierung des Interfaces zurück, falls ihr eine solche bekannt ist. In objektorientierten Programmiersprachen, welche das Interfacekonzept unterstützen und die Möglichkeit bieten, zur Laufzeit Typinformationen zu ermitteln, kann direkt der Typ der Schnittstelle verwandt werden. Dies ist in dieser Form beispielsweise in Java oder C# möglich. Sollte eine Sprache nicht über diese Möglichkeiten verfügen, so stellt dies dennoch keine Einschränkung an das Konzept dar. Als Schnittstellenbeschreibung könnte auch eine Definition auf Basis der CORBA-IDL oder der WSDL[33] dienen. Eine solche Änderung hätte zwar Einfluss auf die interne Realisierung der Vermittlerinstanz, würde das Konzept aber unverändert lassen. Wenn im Folgenden Schnittstellen diskutiert werden, werden diese in Form von UML-Diagrammen dargestellt, da diese Notation eine kompakte und realisierungsunabhängige Darstellung der Sachverhalte ermöglicht.

[33] Sowohl die CORBA Interface Definition Language (IDL) als auch die Web Service Description Language (WSDL) sind Sprachen zur Beschreibung von Schnittstellen. Beide sind unabhängig von einer konkreten Programmiersprache.

Als Nutzer der bereitgestellten Funktionalität kommt zum einen das Programmiermodell in Frage, aber auch die Realisierungen anderer Dienste oder Teile des Frameworks können Nutzer eines Dienstes sein. Damit die bereitgestellten Funktionalität genutzt werden kann, muss zunächst für alle potenziellen Nutzer ein Zugriff auf die Vermittlungsinstanz möglich sein. Zudem muss sichergestellt werden, dass alle Klienten immer auf ein und dieselbe Vermittlerinstanz zugreifen. Hierbei handelt es sich um das typische Einsatzszenario für das so genannte Singleton-Muster [Gam95]. Dieses Entwurfsmuster adressiert das Problem, einen „*global and single point of access*“ [Lar02] innerhalb einer Anwendung bereitzustellen.

Um über die reine Verzeichnisfunktion hinaus eine Verwaltung der Servicerealisierungen durch die Vermittlerinstanz zu ermöglichen, wird ein Basisinterface `IService` vorgegeben. Jedes Serviceinterface muss von diesem abgeleitet sein, jede Servicerealisierung stellt also zumindest die dort deklarierten Methoden bereit. Diese dienen dazu, der Realisierung eines Services die Möglichkeit sowohl zur Initialisierung vor der ersten Verwendung als auch zur Ausführung von eventuell für eine ordnungsgemäße Beendigung notwendigen Aufgaben zu geben. Die Methode `InitializeService()` wird von der Registry beim Hinzufügen eines Services aufgerufen, die Methode `UnloadService()` vor dem Entfernen aus dem Bestand.

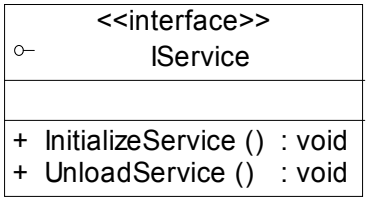


Abb. 4.4: Das Interface IService[34]

Da eine der Anforderungen an das Framework die einfache Verwendbarkeit durch spätere Nutzer ist, deklariert das in Abb. 4.4 gezeigte Interface `IService` nur die notwendigen Methoden für einen minimalen Funktionsumfang. Das IDE-Framework soll nicht nur wieder verwendbare Funktionalität bereitstellen, sondern auch offen für Anpassungen und Erweiterungen sein. Eine Erweiterung der Basisfunktionalitäten oder ein Ersatz bestehender Funktionalitäten des Frameworks erfordert die Implementation neuer Services. Ein überfrachtetes Interface, welches für jedes denkbare Szenario Methoden bereitstellt, erhöht die Hürde zur Implementation neuer Services und erschwert das Verständnis.

Ein einzelner Service wird (vgl. hierzu Abschnitt 3.3) nicht notwendig durch eine Komponente realisiert, vielmehr kann eine Komponente auch mehrere Dienste anbieten. Wenn eine solche Realisierung gewählt wird, sind die Nachteile, welche durch die Kapselung mehrerer Dienste in einer Komponente entstehen, zu beachten. Es ist dann nicht mehr möglich, diese Dienste einzeln zu ersetzen oder in einem anderen Zusammenhang wieder zu verwenden. Szenarien, in denen dieses Vorgehen sinnvoll ist, ergeben sich beispielsweise, wenn zu einem Dienst spezialisierte Dienste durch Ableiten aus Basisklassen des allgemeineren Dienstes erzeugt werden. Auch in Fällen, in denen Dienste zwingend das Vorhandensein anderer Dienste benötigen, kann es sinnvoll sein, diese als Verteilungseinheit zusammenzufassen. Theoretisch sollte durch das gemeinsame Laden der Dienste in

[34] Das Icon zur Verdeutlichung der Schnittstelle ist eine Eigenheit des verwendeten UML-Modellierungswerkzeugs (Sybase PowerDesigner 11 [Syb06]). Die UML lässt explizit die Verwendung von Icons als Stereotypen zu, da die Verwendung der textuellen Variante jedoch gebräuchlicher ist, wurde sie hier ergänzt.

einer Komponente ein Geschwindigkeitsvorteil für die Anwendung entstehen, in den prototypischen Realisierungen ließ sich dies jedoch nicht bestätigen.

Die Instanzierung der Komponenten und die Bereitstellung der Dienste durch das Framework erfolgt beim Laden der Anwendung. Je nach verwendetem Komponentenmodell müssen die Komponenten, die einen oder mehrere Dienste bereitstellen, dem Framework eventuell vor dem Start der Anwendung bekannt gemacht werden. Dies kann beispielsweise durch entsprechende Einträge in einer Datei geschehen. Im Falle der umgesetzten prototypischen Implementierung reicht das Bereitstellen in einem speziellen Verzeichnis (Details einer Realisierung zum Laden und Instanzieren von Komponenten aus einem Verzeichnis werden in Kapitel 6, ab Abschnitt 6.2 besprochen). Unter Verwendung von Reflexion werden alle im Verzeichnis enthaltenen Komponenten ermittelt, die das Interface `IService` realisieren. Diese werden instanziert und die Dienste zur Registry hinzugefügt.

Das Laden und Instanzieren der Komponenten, die die Dienste bereitstellen, ist abhängig von dem zur Realisierung des hier vorgestellten Entwurfs verwendeten Komponentenmodell. Diese Funktionalität kann jedoch getrennt von der eigentlichen Verwaltung der Dienste realisiert werden, so dass eine Anpassung (z. B. durch die Verwendung eines anderen Komponentenmodells) keine grundlegenden Änderungen des Entwurfs erfordert. Zudem wird die Funktionalität zum Laden und Instanzieren von Komponenten auch von dem in Abschnitt 4.3.5, Dynamische Interaktionskomponenten, besprochenen Ansatz benötigt. Dies ist ein zusätzlicher Grund, der für die Kapselung der zum Laden und Instanzieren von Komponenten benötigten Funktionalität in Form einer eigenständigen Komponente innerhalb des Frameworks spricht. Dies erleichtert zum einen die Austauschbarkeit und erhöht zum anderen die Wiederverwendbarkeit.

Da im Rahmen der hier diskutierten Architektur gemeinsam nutzbare Funktionalität innerhalb des Frameworks als Service bereitgestellt wird, liegt es nahe, dass auch die eben angesprochene Komponente ihre Funktionalität in dieser Form zur Verfügung stellt. Dieser Dienst zum Laden von Komponenten kann dann jedoch aus nahe liegenden Gründen nicht wie der Rest der Dienste geladen und instanziert werden.

Generell können durch die gegenseitige Verwendung Abhängigkeiten zwischen den Diensten auftreten. Dies ist zum einen bei der Realisierung der Dienste zu berücksichtigen. Insbesondere können auch Fehlersituationen auftreten, in denen ein benötigter Dienst nicht verfügbar ist. Dienste, die als Klienten auftreten, müssen dementsprechend defensiv entworfen werden. Zum anderen müssen die Abhängigkeiten zwischen den Diensten auch bei der Implementation des Framework-Kerns, welcher die für den Start der Applikation nötigen Funktionalitäten realisiert, berücksichtigt werden. Die Funktionalität des Kerns umfasst im Wesentlichen die Aufrufe zum Starten der Teilsysteme des Frameworks. Über diese Funktionalität hinaus stellt der Anwendungskern das Hauptfenster der Anwendung bereit.

Die unten stehende Abb. 4.5 verdeutlicht den Zusammenhang der bisher besprochenen Teilsysteme bzw. Komponenten einer auf dem Framework basierenden Anwendung. Der „Anwendungskern“ nutzt die „ServiceVerwaltung“; beide sind als Pakete dargestellt, da es sich um Anwendungsteile, aber nicht notwendigerweise um Komponenten handelt. Das Interface `IService` wird wie angesprochen in einer eigenen Komponente „IServiceInterface“ gekapselt, ebenso der Service zum Laden und Instanzieren von Komponenten. Da letzterer als Dienst das Interface `IService` implementieren muss, ist die Komponente „KomponentenLaden“ abhängig von der Interfacekomponente.

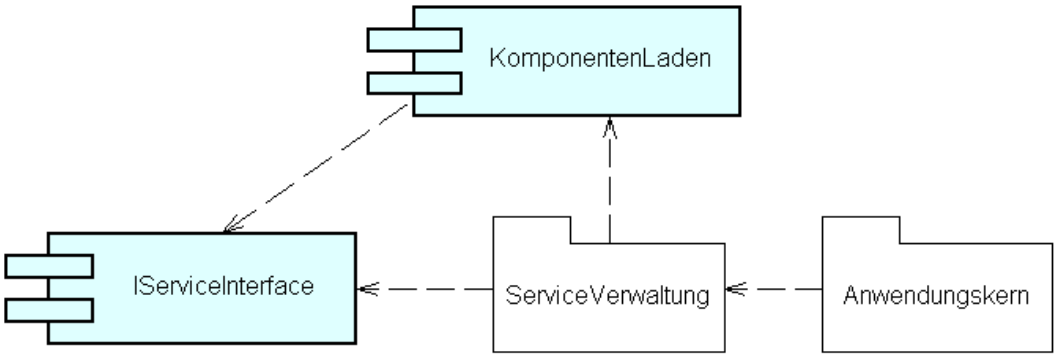


Abb. 4.5: Überblicksweise Darstellung des Zusammenhangs zwischen den bisher besprochenen Teilsystemen bzw. Komponenten des Cobamos-Frameworks

Ein interessanter Aspekt im Zusammenhang mit der Kapselung von Funktionalität in Komponenten ist der Umgang mit Anwendungsfehlern (vgl. Abschnitt 8.2, hier wird der Umgang mit logischen Fehlern in einem begrenzten Anwendunsszenario diskutiert), in diesem speziellen Fall mit Fehlern in den Services des Frameworks. Es ist wahrscheinlich, dass bei einem Einsatz des Frameworks Komponenten unterschiedlicher Entwickler in der resultierenden Anwendung integriert sind. Prinzipiell können in jeder der Komponenten und im Framework selbst Fehler auftreten. Fehler, die nur lokal begrenzte Auswirkungen haben und bei denen eine Behandlung innerhalb der jeweiligen Komponente bzw. des betroffenen Anwendungsteils möglich ist, ohne dass Benutzerinteraktion notwendig wäre, können auch lokal behandelt werden. Aber schon wenn sich die Notwendigkeit einer Interaktion mit dem menschlichen Nutzer ergibt, ist dies nicht mehr sinnvoll. Falls jeder Entwickler eigene Vorstellungen der Fenstergestaltung etc. bei den Fehlermeldungen umsetzt, erscheint die Anwendung für den Nutzer nicht mehr geschlossen[35]. Wenn jede Komponente eine eigene Strategie für das Protokollieren von Fehlern verfolgt, ist im Falle einer notwendigen Wartung unter Umständen eine Vielzahl von Dateien und Formaten zu prüfen. Zudem ist es nicht zu erwarten, dass in einer Komponente, die einen Dienst nutzt angemessen auf einen Anwendungsfehler in der Dienstrealisierung reagiert werden kann. Daher ist es sinnvoll eine frameworkweite, konsistente Strategie zur Fehlerbehandlung festzulegen. In den Fällen, in denen eine lokale Fehlerbehandlung nicht möglich ist, sollte im Falle eines Anwendungsfehlers eine Ausnahme (Exception) ausgelöst werden. Diese können durch das Framework abgefangen und behandelt werden. Dadurch ergibt sich eine Anforderung an die Implemetationssprache für das Framework, sie muss Exceptions als Sprachmittel bereitstellen. Eine Vereinheitlichung der Protokollierung kann leicht durch das Bereitstellen eines entsprechenden Dienstes, der diese Funktionalität anbietet, erreicht werden.

Aus diesem Grund sind in den weiter oben gezeigten Signaturen der Methoden des Interfaces `IService` auch keine Status- oder Rückgabevariablen vorgesehen. Dieses Vorgehen zur Fehlerbehandlung führt bei der Realisierung von Frameworks schnell zu inkonsistenten und schwierig zu nutzenden Schnittstellen. Die so genannte Win32 API (Application Programming Interface), ein zur Nutzung mit der Programmiersprache C++ vorgesehenes Framework zur Programmierung in den Betriebssystemen der Microsoft Windows Familie, ist hier ein gutes Beispiel. In diesem Framework werden Rückgabewerte in unterschiedlichster Form zur Fehlerbehandlung verwendet, ein Umstand der die Nutzung deutlich erschwert[36] [CA06].

[35] Diese einheitliche Gestaltung der Interaktionsmöglichkeiten ist natürlich nicht nur im Zusammenhang mit der Fehlerbehandlung von Bedeutung, sondern für die gesamte Anwendung. Dieser Punkt wird in 4.3.4 aufgegriffen.

[36] In [CA06] werden beispielsweise die Nutzung von booleschen Werten und der Datenstruktur HRESULT – beide werden von Methoden zur Rückgabe von Statuswerten verwendet – angeführt. Andere Teile dieser API

4.3.4 Dienste als Integrationspunkt

Neben der oben beschriebenen Nutzung der Dienste zur Wiederverwendung wird über diese Architektur ein Anknüpfungspunkt für das Programmiermodell innerhalb der Entwicklungsumgebung geschaffen. Dieser ist beispielsweise notwendig, um eine Integration in die Nutzungsoberfläche zu erlauben.

Im vorangegangenen Abschnitt wurde bereits im Zusammenhang mit der Fehlerbehandlung die Notwendigkeit eines geschlossenen äußeren Erscheinungsbildes der auf dem Framework basierenden Anwendung angesprochen. Vergleichbares gilt für die gesamte Gestaltung der Nutzungsoberfläche und der Interaktionsmöglichkeiten der Anwendung. Dies beinhaltet zum Beispiel die Gestaltung von Meldungsfenstern, aber auch Aspekte wie die Nutzung der Statuszeile oder der Titelleiste des Hauptfensters der Anwendung fallen in den Bereich dieser Betrachtungen.

Die einheitliche Gestaltung der Benutzerinteraktionen – hierunter fällt neben der grafischen Gestaltung auch die Interaktionslogik – ließe sich auch durch die Vorgabe von entsprechenden Richtlinien erreichen. Jede für das Framework entwickelte Komponente müsste diese dann umsetzen. Zum einen wird eine solche Verpflichtung von Softwareentwicklern vielfach als Beschränkung der individuellen Kreativität gesehen und dementsprechend auch umgangen. Zum anderen würde eine Änderung an den Vorgaben, beispielsweise zur Anpassung der Anwendung auf eine andere Zielgruppe, notwendige Änderungen an den bereits fertig gestellten Komponenten nach sich ziehen. Daher ist dieses Vorgehen für eine Anwendung, in der diese Art von Anpassbarkeit gefordert wird, nicht geeignet.

Alternativ kann eine Klassenbibliothek bereitgestellt werden, welche Interaktionselemente wie Dialoge und Ähnliches bereitstellt. Für einfache Dialoge ist dies ein akzeptabler Lösungsansatz. Bei Änderungen an der Bibliothek wird aber unter Umständen – selbst bei identischen Interfaces der Klassenbibliothek – ein erneutes Übersetzen der Komponenten nötig. Darüber hinaus stellt dieser Ansatz keine Lösung für die Nutzung von Elementen des Hauptfensters wie der oben angesprochen Statuszeile dar.

Die im vorherigen Abschnitt dargestellte servicebasierte Anwendungsarchitektur bietet sich als Lösung für diese Problematik an. Die eigentliche Interaktion mit dem menschlichen Nutzer der Anwendung wird durch einen entsprechenden Dienst realisiert; der Nutzer des Dienstes teilt dann beim Aufruf nur mit, um welche Art der Interaktion (z. B. Information, Frage oder Fehlermeldung) es sich handelt und übergibt gegebenenfalls anzuzeigenden Text als Parameter an den Dienst. Die Art der Präsentation wird durch den Dienst bestimmt. Eine Anpassung kann – unter der Voraussetzung, dass die Schnittstelle des Dienstes Bestand hat – durch ein Austauschen der den Dienst realisierenden Komponente vorgenommen werden und erfordert keine Änderungen an den bereits fertig gestellten Komponenten, die diesen Dienst nutzen.

Diese Lösung bietet gegenüber den Designrichtlinien den Vorteil der Wiederverwendung und damit einer Entlastung von Entwicklern in zukünftigen Projekten auf dieser Basis. Darüber hinaus sind Änderungen an einer einzigen Stelle, der Realisierung des Dienstes, möglich. Auch hier gilt, dass diese Änderung im Falle unveränderter Schnittstellen kein Anpassen der Nutzer nach sich zieht (lokale Änderbarkeit). Entwickler zukünftiger Komponenten für das Framework könnten den ihnen durch den Funktionsumfang der Dienste vorgegebenen Rahmen als einschränkend empfinden. In diesem Fall ist es jedoch möglich, eine neue Version des Dienstes mit erweiterter Funktionalität zu entwickeln. Falls eine Erweiterung der Schnittstelle des Dienstes notwendig ist, kann diese ebenfalls unter bestimmten Bedingungen so erfolgen, dass bereits vorhandene Dienstnutzer nicht angepasst werden müssen.

---

erfordern vom Nutzer den Aufruf einer Methode GetLastError zum Abfragen von Fehlern.

Hierzu ist es erforderlich, dass die neue Version der Schnittstelle von der vorherigen abgeleitet wird. In der Folge kann die Dienstrealisierung von den neuen Nutzern mit der erweiterten Funktionalität über die neuere Schnittstelle genutzt werden, die zum Zeitpunkt der Anpassung bereits vorhandenen Nutzer können die Funktionalität weiterhin über die ihnen bekannte Schnittstelle nutzen.

Zum Zugriff auf Elemente der grafischen Benutzeroberfläche des Hauptfensters können ebenfalls Dienste verwendet werden. Diese sollte sich aber nicht an einem konkreten Oberflächenelement orientieren. Dies wäre zum Beispiel der Fall, wenn ein Dienst eine Referenz auf das jeweilige Element an seine Nutzer liefert. Vielmehr sollte sich die Schnittstelle des Dienstes an den gängigen Aufgaben, die unter Verwendung der Oberflächenelemente durchgeführt werden sollen, orientieren. Für den Dienst zum Zugriff auf die Statusleiste darf also keine Methode `GetStatusbar()` existieren, die eine Referenz auf das konkrete Objekt liefert. Diese Form der Implementation würde Wissen über die konkrete Realisierung der Statusleiste in den Nutzern erfordern (dies schlägt sich in Form von Aufrufen der Art `GetStatusbar().Panels[0].Text = text` nieder). Stattdessen wird beispielsweise in der Schnittstelle des Dienstes eine Methode `DisplayText(string text)` bereitgestellt. Dies hat den Vorteil, dass eine Änderung der Anwendungsoberfläche die Schnittstelle des Dienstes unverändert lässt und damit die Nutzer nicht von den Änderungen betroffen sind. Im Verlauf der prototypischen Umsetzung dieses Konzeptes wurde nach der Fertigstellung des Frameworks eine Umstellung auf eine andere Bibliothek mit Oberflächenkomponenten vorgenommen. Die Schnittstellen und Typen dieser Bibliothek waren mit denen der zuvor verwendeten inkompatibel. Durch die eben angesprochene Ausrichtung der Dienstschnittstellen an den Aufgaben konnten die Änderungen jedoch in den zuständigen Diensten abgefangen werden, ohne dass eine Änderung an deren Schnittstellen notwendig war. Die den Dienst nutzenden Anwendungsteile mussten daher nicht verändert werden.

4.3.5 Dynamische Interaktionskomponenten

Der oben vorgestellte serviceorientierte Ansatz zielt darauf ab, Funktionalität für andere Objekte, Dienste oder Komponenten innerhalb einer Softwareapplikation bereitzustellen. Dabei ist nicht vorgesehen, dass ein Dienst mit einem menschlichen Nutzer der Anwendung interagiert. Es gibt jedoch eine Reihe von denkbaren Szenarien, in denen eine Erweiterung oder Anpassung der Interaktionsmöglichkeiten der Entwicklungsumgebung notwendig wird. Dazu gehört beispielsweise die Änderung der vorhandenen Nutzerunterstützung zur Anpassung auf eine andere Zielgruppe, so kann für unerfahrenere Anwender an geeigneten Stellen ein interaktives Tutorium in die Anwendung integriert werden. Erfahrenere Anwender fühlen sich durch zu viel Unterstützung unter Umständen in ihrer Arbeit behindert. Eventuell soll die Funktionalität zum Speichern von Daten des Programmiermodells angepasst werden oder eine zusätzliche Möglichkeit dazu bereitgestellt werden. In diesen Fällen reicht eine Realisierung über einen Service nicht aus, da dem Nutzer die Möglichkeit zur Interaktion gegeben werden muss.

Um diese Interaktion mit einem menschlichen Nutzer zu ermöglichen, muss die bereitgestellte Funktionalität über die grafische Nutzungsoberfläche (GUI, Graphical User Interface[37]) der Enwicklungsumgebung ansprechbar sein. Die Integration in diese muss so erfolgen, dass die Anwendung in sich geschlossen wirkt. Die bereitgestellte Funktionalität muss sich also beispielsweise über die vorhandenen Menüs oder Werkzeugleisten der Anwendung aktivieren lassen. Auch die grundlegenden Interaktionsmuster vergleichbarer Aufgaben sollten sich entsprechen. Eine solch „nahtlose“ Integration ist wichtig, da anderenfalls ein Nutzer gezwungen wird, sich fortwährend die

[37] Auch in der deutschsprachigen Literatur wird häufig die englische Abkürzung verwandt (siehe z. B. [Bal05]), diese scheint inzwischen gebräuchlicher zu sein als BO Benutzungsoberfläche.

zur Erledigung seiner Aufgaben notwendigen Schritte neu zu erschließen. Diese für die Nutzungsoberfläche angestrebte Eigenschaft wird als Erwartungskonformität bezeichnet und ist einer der Grundsätze für die Gestaltung von Interaktionsoberflächen [Bal05].

Um diese Anforderungen zu erfüllen, wurde ein Ansatz gewählt, welcher auf der Kapselung von durch den Nutzer aktivierbarer Funktionalität in Komponenten basiert. Eine solche funktionale Entität wird im Folgenden als Kommandokomponente oder kurz als Kommando bezeichnet. Analog zum Umgang mit den Realisierungen der Dienste in dem zuvor besprochenen serviceorientierten Teil der IDE-Architektur werden auch diese Kommandokomponenten erst zur Laufzeit der Anwendung eingebunden.

Die zugrunde liegende Idee entspricht im Prinzip einer Übertragung des in der objektorientierten Entwicklung bekannten Command-Musters [Gam95] auf Komponenten. Dieses Verhaltensmuster basiert auf der Idee, Operationen in Objekten zu kapseln. Eingesetzt wird dieses Muster beispielsweise, um widerrufbare Aktionen („Undo“) oder Warteschlangen von Aktionen zu realisieren. In Abb. 4.6 ist die grundlegende Struktur dieses Entwurfsmusters dargestellt. Die Definition einer gemeinsamen abstrakten Oberklasse ermöglicht es, unterschiedliche, konkrete Command-Objekte in identischer Art und Weise zu nutzen.

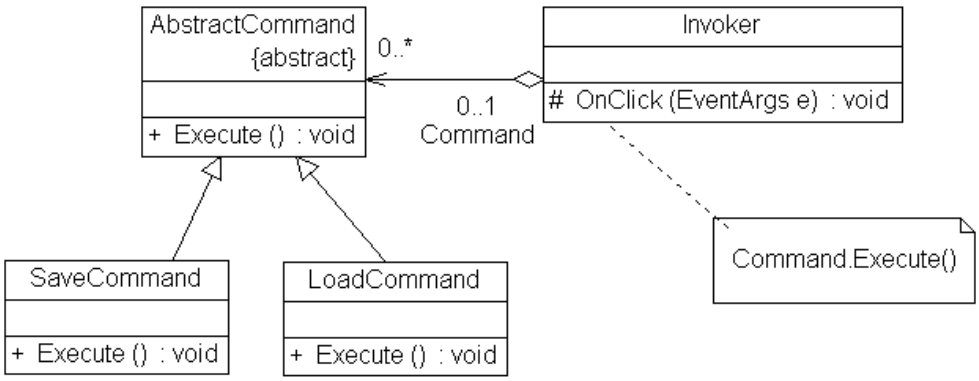


Abb. 4.6: Das Command-Muster

Durch die Kombination dieses Musters mit der Möglichkeit des dynamischen Ladens von Komponenten können auch Kommandos in eine Anwendung integriert werden, die zur Entwurfszeit nicht vorgesehen waren. Jedes dieser Kommandos verfügt über eine Aktion, diese kann ggf. auch komplexer sein, sollte aber „in sich abgeschlossen“ sein. Über eine Interaktion mit der grafischen Nutzungsoberfläche des Systems kann eine solche Aktion durch den Nutzer angestoßen werden.

Um neben dem reinen Laden der gekapselten Funktionalität auch eine Integration in die Nutzungsoberfläche der Anwendung zu ermöglichen, sind Erweiterungen zum ursprünglichen Command-Muster notwendig. Zu diesem Zweck wird eine Schnittstelle `ICobamosCommand` definiert, über welche zusätzliche Informationen abgefragt werden können.

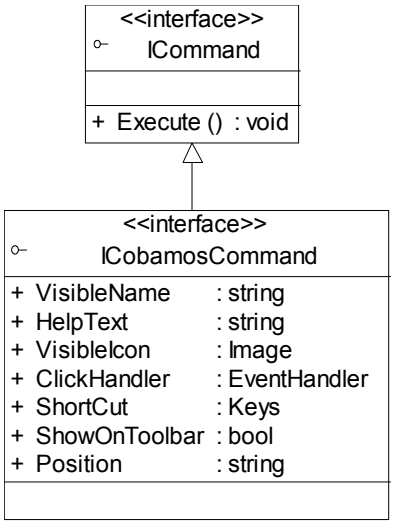


Abb. 4.7: Das Interface ICobamosCommand

In erster Linie handelt es sich dabei um Informationen, welche für die Darstellung von Interaktionsmöglichkeiten benötigt werden, mit denen die Funktionalität eines solchen Kommandos angesprochen werden soll. Die hier gezeigte Auswahl dieser Informationen basiert dabei auf dem in der prototypischen Implementation umgesetzten Funktionsumfang, hier sind durchaus Erweiterungen denkbar. Im erstellten Prototypen des IDE-Frameworks können diese Kommandos über Schaltflächen in einem Menü (die Bestimmung der Position erfolgt über die Daten der Zeichenkette `Position`) oder optional zusätzlich auf einer so genannten Werkzeugleiste aktiviert werden. Neben einer Beschriftung für die Schaltfläche kann ein optionales Icon angegeben werden.

Für die Verwaltung der Kommandokomponenten steht ein eigenes Teilsystem im Framework zur Verfügung. Dieses lädt die Komponenten beim Anwendungsstart, fragt für jede die Informationen zur Darstellung in der Nutzungsoberfläche der Anwendung ab, erzeugt die entsprechenden Interaktionsmöglichkeiten und integriert diese in die Nutzungsoberfläche. Weitere Details zur prototypischen Realisierung dieser Funktionalität finden sich im Abschnitt 6.3.

Da die Instanzen der die Kommandos realisierenden Komponenten ebenfalls über die Serviceregistry Zugriff auf die von dem IDE-Framework bereitgestellten Dienste haben, kann zum einen die hier vorhandene Funktionalität genutzt werden. Zum anderen können die Dienste genutzt werden, die Benutzerinteraktionen und Zugriff auf die Nutzungsoberfläche der Entwicklungsumgebung ermöglichen, so dass eine bessere visuelle Integration der Kommandos in die Anwendung erreicht werden kann.

Der Zugriff auf das Dienstesystem des Frameworks ermöglicht es beispielsweise, visuelle Oberflächen für die Steuerung oder Konfiguration von Diensten durch den Nutzer der Applikation zu realisieren. Eine solche würde als Kommando in einer separaten Komponente realisiert, durch die in der Anwendung erzeugten Interaktionselemente kann diese vom Nutzer angesprochen werden. Dieses Vorgehen erzwingt eine Trennung von grafischer Nutzungsoberfläche und Funktionalität, ein Merkmal eines guten Entwurfs.

Die Flexibilität des bisher vorgestellten Ansatzes zeigt sich deutlich an dem prototypisch realisierten Kommando zur Speicherung von Modellzuständen. Da eine Festlegung auf ein bestimmtes Programmiermodell vermieden werden soll, ist es kaum möglich, ein allgemeines und in jedem Fall geeignetes

Verfahren zur Speicherung zu entwerfen[38]. Folglich ist es nahe liegend, die Speicherung der Modelldaten in eine Kommandokomponente zu kapseln und bei einem Wechsel des Programmiermodells auch das zum Speichern notwendige Kommando zu wechseln. An dieser Stelle tritt jedoch das Problem auf, dass das Kommando durch das Framework geladen und von diesem gegebenenfalls aufgerufen wird (inversion of control), das Kommando hat aber keinen Zugriff auf Teile des Frameworks oder das Programmiermodell. Dieses Problem ist durch die Realisierung eines Dienstes zum Zugriff auf das Modell zu lösen. Damit steht ein applikationsweit verfügbarer Zugriffspunkt auf die Daten des Modells bereit. Über diesen kann das Speicherkommando ebenfalls auf die Daten zugreifen, ein Kommando zum Laden der Daten kann hier die Daten in das Programmiermodell zurückschreiben. Darüber hinaus ist es wahrscheinlich, dass mit einem Wechsel des Programmiermodells auch eine Änderung der Zugriffe über den Dienst einhergehen muss. Da dieser durch eine eigenständige Komponente realisiert wird, ist eine solche Anpassung jedoch problemlos möglich. Eine detailliertere Betrachtung der Implementation des Dienstes zum Zugriff auf das Modell findet sich in auf Seite 103 in Kapitel 6, das Kommando zum Speichern des Modellzustandes wird in Abschnitt 6.6 eingehender behandelt.

Hier ist anzumerken, dass durch die im Framework vorhandene Flexibilität im Hinblick auf den Austausch von Dienstrealisierungen und Kommandokomponenten Fehlersituationen auftreten können, in denen der von einem Kommando benötigte Dienst nicht vorhanden ist. Dies ist analog zu den Abhängigkeiten zwischen den Diensten zu sehen. Prinzipiell ist eine Prüfung der von einem Kommando benötigten Ausführungsumgebung durch das Framework denkbar. Hierfür müsste aber ein entsprechender Mechanismus geschaffen werden, über den die Kommandokomponenten ihre Anforderungen kommunizieren können. Die Beschreibung solcher Anforderungen führt jedoch schnell in den Bereich der Spezifikation von Semantik; die Entwicklung einer befriedigenden Lösung hierfür ist im Rahmen dieser Arbeit nicht zu leisten. Bei der Realisierung des Frameworks wird daher der Ansatz verfolgt, dass der Entwickler einer Komponente detaillierte Kenntnisse über den von der Komponente benötigten Kontext besitzt. Eine eventuell erforderliche Prüfung der Ausführungsumgebung, z. B. ob benötigte Dienste vorhanden sind, ist bei den Kommandokomponenten anzusiedeln. Falls hierdurch eine Kontextabhängigkeit ensteht – wie es beispielsweise der Fall wäre, wenn die Funktionalität der Komponente beim Fehlen eines Dienstes deaktiviert würde – so muss diese ausreichend dokumentiert werden. Anderenfalls wäre die in Kapitel 3 aufgestellte Forderung, dass eine Komponente nur explizite Kontextabhängigkeiten besitzen darf, verletzt.

4.4 Entwurf eines austauschbaren Programmiermodells

In diesem Abschnitt wird die Realisierung des bisher als Black-Box betrachteten Programmiermodells für das entworfene IDE-Framework erläutert. Zunächst wird die Bedeutung der Modellbildung für die Softwareentwicklung und die Endnutzerprogrammierung im Besonderen sowie der Begriff des Programmiermodells genauer erläutert.

4.4.1 Modelle im Kontext der Softwareerstellung

Bevor es möglich ist, eine vielseitig verwendbare oder anpassbare Realisierung eines Programmiermodells für die Endnutzerprogrammierung zu entwerfen, müssen die grundsätzlichen Anforderungen

[38] Ein sehr allgemein verwendbarer Ansatz lässt sich auf Basis objektorientierter Datenbanken realisieren, dies wurde auch prototypisch umgesetzt. Dieses Vorgehen bedingt allerdings gewisse Anforderungen an die Realisierung des Programmiermodells, Näheres hierzu ist bei den Details der Realisierung in 6.6 zu finden.

an solche Modelle erhoben werden. Es ist daher nahe liegend, sich zunächst vor Augen zu führen, wie Modelle in diesem Umfeld gebraucht werden und welche Aufgaben unterstützt werden müssen.

Der Prozess des Erstellens einer Softwareapplikation beinhaltet immer in der einen oder anderen Form das Erstellen eines Modells. Dies ist wenig verwunderlich, da das angestrebte Endergebnis, die Anwendung, selbst ein Modell der Realität ist (oder sein sollte). Im Verlauf der Softwareentwicklung werden in der Regel unterschiedliche Modelle erstellt. Zum einen werden während der Anforderungsanalyse Modelle erstellt, welche den darzustellenden Sachverhalt bzw. die Realität abbilden. Zum anderen gibt es Modelle, welche das zukünftige Softwaresystem beschreiben, diese entstehen in den späteren Phasen des Entwicklungsprozesses. Ein Sachverhalt bzw. ein System kann dabei unter verschiedenen Gesichtspunkten und auf unterschiedlichen Abstraktionsebenen dargestellt werden; ab einer gewissen Komplexität ist dieses Vorgehen unerlässlich.

Für ein Softwaresystem zur Kundenbetreuung existieren dann beispielsweise beim Einsatz der UML 2.0 als Modellierungssprache neben einem Paketdiagramm, welches statische Zusammenhänge auf einer sehr hohen Ebene beschreibt, auch Aktivitätsdiagramme zur Beschreibung des Systemverhaltens auf einer abstrakten Ebene. Zur verfeinerten Darstellung des Systemverhaltens werden Sequenz- oder Kommunikationsdiagramme verwendet, die statische Struktur des Systems auf einer Ebene mit höherem Detaillierungsgrad wird durch Klassendiagramme repräsentiert.

Die bisherigen Ausführungen machen bereits deutlich, dass im Zusammenhang mit der Entwicklung von Software unterschiedliche Typen von Modellen relevant sein können. Der Typ eines Modells wird im Allgemeinen als Metamodell bezeichnet (vgl. beispielsweise [Per02]). Diese Beziehung kann analog zum aus der Objektorientierung bekannten Konzept von Klasse und konkreter Instanz gesehen werden. Ein Modell ist dementsprechend eine Instanz eines Metamodells. Während das Modell ein System beschreibt, beschreibt das Metamodell Modelle. Das Ziel der Metamodellbildung ist es, von konkreten zu modellierenden Objekten und Beziehungen zu abstrahieren und stattdessen deren gemeinsame Eigenschaften zu beschreiben. Die unten stehende Abb. 4.8 verdeutlicht diesen Zusammenhang.

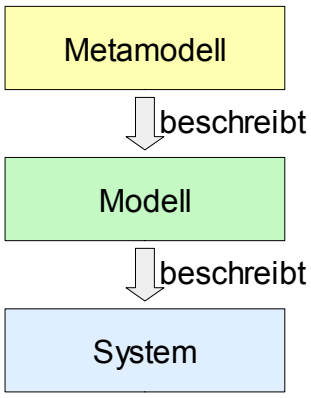


Abb. 4.8: Beziehung von Metamodell und Modell

Auch im Hinblick auf die Endnutzerprogrammierung können unterschiedliche Metamodelle zum Entwurf einer Anwendung Verwendung finden. Die von der in Kapitel 2 auf Seite 27 angesprochenen Software Stagecast Creator präsentierten Darstellungen fußen augenscheinlich auf einem regelbasierten Modell; der im weiteren Verlauf dieser Arbeit (siehe Kapitel 7, 8 und 9) umgesetzte Ansatz lässt sich am ehesten als „flussbasiert“ umschreiben. Auch aufgrund der bisher am Institut für Informatik der Johannes Gutenberg-Universität Mainz durchgeführten Projekte, welche an das Gebiet

der Endnutzerprogrammierung angrenzen und die vorliegende Arbeit motiviert und beeinflusst haben, wird deutlich, dass die verwendeten Modelle sehr unterschiedlich sein können. So werden im Rahmen der Arbeit von Reinhardt [Rei00] Zustands-Ereignis-Graphen (ZE-Graphen) verwendet. Andere Arbeiten, welche Modellgeneratoren im Bereich der Sportinformatik zum Thema hatten, basierten auf Level-Raten-Modellen (eine Erläuterung zu Level-Raten-Modellen findet sich in [Per02]). Dies lässt bereits die große Breite an Modellen erahnen, die für den Bereich der Endnutzerprogrammierung in Frage kommen. Diese Breite ist es auch, die ein sinnvolles Abstrahieren auf ein gemeinsames Metamodell für all diese Ansätze unrealistisch werden lässt. Ein solcher Ansatz muss entweder Beschränkungen der Modelle beinhalten oder aber so allgemein werden, dass die resultierende Komplexität des Metamodells eine Verwendung stark erschwert.

Für die hier betrachtete Realisierung bedeutet dieses, dass der Entwurf nach Möglichkeit keine Festlegung auf ein konkretes Metamodell implizieren sollte. Dies lässt sich – auf Basis des bisherigen Entwurfs ist dieses Vorgehen nahe liegend – durch Kapselung des Modells in Form einer Komponente realisieren. Das Framework muss geeignete Integrationspunkte für das Modell bereitstellen und die Möglichkeit zum Laden und Instanzieren eines solcherart gekapselten Modells bieten.

Es wäre auch ein Ansatz denkbar, der eine interne Realisierung zur Speicherung eines Modells vorschreibt[39] und diese als festen Teil des Frameworks realisiert. Eine solche Festlegung bedeutet in der Folge jedoch, dass jedes in Frage kommende Modell auf diese Speicherstrukturen abbildbar sein muss. Auf der Ebene der Modelle heißt dies, dass eine Modelltransformation zwischen einer Instanz des für die Endnutzerprogrammierung verwendeten Metamodells und dem gewählten Speichermodell existieren muss. Diese Transformation muss umkehrbar sein, da im Falle einer solchen Realisierung Aktionen des Frameworks auf dem internen Modell durchgeführt werden (beispielsweise Speichern und Laden von Modellinstanzen). Solche Änderungen der Modelldaten müssen auch im externen Modell reflektiert werden. Da dem Nutzer der Anwendung auch eine Sicht auf das externe Modell gezeigt werden muss, und hier vorgenommene Veränderungen auch auf das Modell übertragen werden müssen, ergibt sich die in Abb. 4.9 gezeigte Situation.

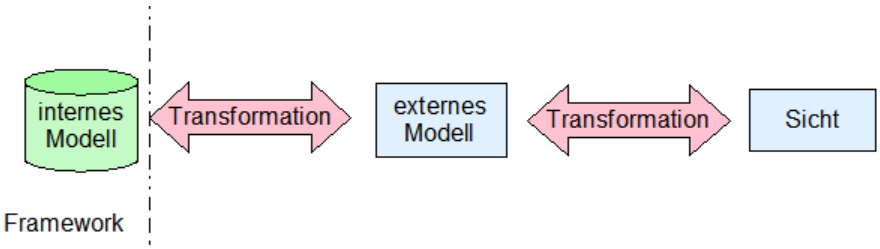


Abb. 4.9: Notwendige Modelltransformationen

Diese Überlegungen machen deutlich, dass dieses Vorgehen drei Ebenen innerhalb des Gesamtmodells erfordert, wobei zwei – das interne und das externe Modell – im Wesentlichen die selbe Aufgabe besitzen. Beide realisieren ein vollständiges Modell, das interne stellt dieses innerhalb der Entwicklungsumgebung bereit, das externe für die Sichten. Zudem ist die Aufgabe, ein internes Modell zu finden, welches eine möglichst große Gruppe potenzieller Metamodelle für das externe Modell erlaubt und umkehrbare Transformationen zu den Modellen dieser Gruppe zulässt, ebenfalls recht komplex.

[39] Dabei könnte es sich beispielsweise um eine Implementation für einen **abstrakten Syntaxbaum** (abstract syntax tree, AST) handeln. Diese finden vielfach Anwendung zur Realisierung der internen Speicherstrukturen von Parsern [Gru00].

Es ergibt sich jedoch kein Vorteil aus diesem Vorgehen, welches den zu erwartenden Aufwand rechtfertigen würde.

Daher soll der zu wählende Ansatz die Kapselung eines Modells – dieses entspricht dann dem externen Modell in der obigen Abb. 4.9 – mit einer oder mehreren zugehörigen Sichten erlauben. Zur Integration in das IDE-Framework muss ein Weg gefunden werden, welcher nicht auf der Verwendung eines zusätzlichen, internen Modells bzw. entsprechenden Vorgaben hinsichtlich der Speicherung von Modelldaten beruht.

Die Unterstützung mehrerer Sichten auf ein Modell kann insbesondere im Kontext der Endnutzerprogrammierung große Vorteile mit sich bringen. Für Endnutzer mit unterschiedlichem Kenntnisstand können auf diese Art und Weise angepasste Sichten zum Entwurf einer Anwendung bereitgestellt werden. Auch im Hinblick auf die Unterstützung eines Entwicklungsprozesses sind unterschiedliche Sichten sinnvoll. Diese ermöglichen es beispielsweise, ein Modell auf unterschiedlichen Abstraktionsebenen zu betrachten. Hierdurch kann die Übersicht und damit das Verständnis für eine Anwendung erhöht werden. Spezielle Repräsentationen – beispielsweise in Berichtsform – können Endnutzern helfen, Fehler im Entwurf der Anwendung schneller zu finden. Diese hier nur kurz angesprochenen Möglichkeiten motivieren den folgenden Abschnitt dieser Arbeit.

Um die Kapselung der Sichten zusammen mit einem Programmiermodell und die Einbindung unterschiedlicher Modelle in das IDE-Framework zu realisieren, wird eine Variation des Model-View-Controller Musters verwendet. Die Überlegungen zu diesem Vorgehen werden im nächsten Abschnitt geschildert.

4.4.2 Von Model-View-Controller zu Model-View-Connector

Das **Model-View-Controller (MVC)** Muster ist ein Entwurfsmuster, welches zum Synonym für die Entkopplung der Nutzungsoberfläche eines Programms von den Fachklassen geworden ist. Dieses Muster wurde bereits 1979 im Zusammenhang mit der Sprache Smalltalk [Ree79] beschrieben. Es eignet sich insbesondere für Situationen, in denen unterschiedliche Sichten auf einen gemeinsamen Datenbestand vonnöten sind. Da die Anwendung dieses Musters die Architektur einer Anwendung im Allgemeinen weitgehender beeinflusst als dies bei anderen Entwurfsmustern der Fall ist, wurde auch der Begriff eines Architekturmusters vorgeschlagen [Bus98].

In Abb. 4.10 ist das MVC-Muster im Überblick dargestellt. Hier wird auch deutlich, dass es im Gegensatz zu anderen Entwurfsmustern i. d. R. nicht auf Ebene der Klassen eines Softwaresystems beschrieben wird, ein Punkt der für die Einordnung als Architekturmuster spricht. Das Muster besteht aus den drei Entitäten View, Controller und Model[40].

[40] Im Folgenden wird für das Modell im MVC-Muster auch weiterhin die englische Schreibweise Model verwandt, da diese auch für die Begriffe View und Controller benutzt wird.

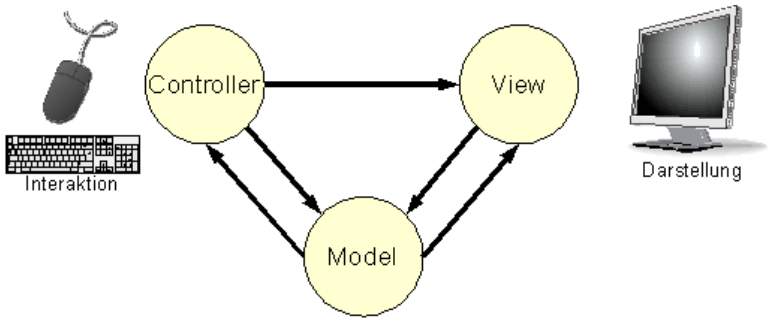


Abb. 4.10: Das Model-View-Controller Muster nach [KP88]

Für das MVC-Muster finden sich in der Literatur voneinander abweichende Darstellungen (siehe z. B. [HH97] [Pot96]), dies betrifft insbesondere die existierenden Abhängigkeiten zwischen den Entitäten bzw. deren Richtungen. In der hier dargestellten Variante ist der Controller für die Verarbeitung der Nutzereingaben zuständig, die View für die Präsentation der Daten und das Model entspricht dem Datenhintergrund. Sowohl der Controller als auch die View sind in der Lage, Nachrichten an das Model zu senden um Änderungen an dessen Daten zu veranlassen. Beide werden über Änderungen an den Daten benachrichtigt. Auch hinsichtlich der Aufgaben, die dem Controller oder der View zuzuordnen sind, finden sich leicht unterschiedliche Interpretationen.

In [KP88] wird bereits die Problematik der Einordnung einzelner Bestandteile visueller Nutzungsoberflächen in dieses Konzept angesprochen. In der Tat ist es so, dass bei einer Betrachtung auf der Ebene einzelner Komponenten[41] der grafischen Nutzungsoberfläche für gewöhnlich die Bestandteile View und Controller zusammengefasst werden und eine Trennung kaum sinnvoll ist. In [Fow06] wird dargelegt, dass die Klassenbibliothek Swing – diese dient in der Programmiersprache Java zur Realisierung visueller Nutzungsoberflächen – zwar ursprünglich auf einem MVC-Ansatz basieren sollte, dies aber aufgrund der engen Kopplung zwischen View und Controller in der endgültigen Realisierung aufgegeben wurde. Diese beiden Teile des Musters wurden zu einem zusammengefasst. Dies entspricht den in den vergleichbaren Klassenbibliotheken von C# und Delphi zu findenden Ansätzen.

Oft werden die Views des MVC-Musters nicht mit einzelnen Komponenten einer grafischen Nutzungsoberfläche sondern mit einem größeren, zusammenhängenden Teil der gesamten Nutzungsoberfläche – im Falle von Desktop-Applikationen beispielsweise mit einem vollständigen Fenster – identifiziert. In solchen Realisierungen kapselt der Controller in der Regel die Geschäftslogik und ist für die Verwaltung der Nutzungsoberfläche (Anzeigen, Entfernen, Aktualisieren und Ähnliches) zuständig. Übertragen auf den ursprünglichen Kontext der Endnutzerprogrammierung bedeutet dies, dass eine Komponente zur Kapselung des Programmiermodells aus dem eigentlichen Modell, einem bis mehreren Fenstern, welche die Sichten auf das Modell realisieren, und einem Controller[42], welcher die Verwaltung der Fenster und die Bereitstellung der auf dem Modell möglichen

[41] Der Begriff visuelle Komponente oder Oberflächenkomponente ist in der Literatur für die einzelnen Elemente einer Nutzungsoberfläche üblich. Daher wird dieser auch im Rahmen dieser Arbeit verwendet, wenn keine Verwechslungsgefahr mit der in Kapitel 3 eingeführten Komponentendefinition besteht. Es ist jedoch zu bemerken, dass Oberflächenkomponenten im Allgemeinen nicht alle der geforderten Eigenschaften einer Komponente im Sinne dieser Definition besitzen.

[42] Die in [HH97] vertretene Sichtweise, dass beim Vorhandensein mehrerer Views zu einem Model in einem MVC-Muster jede View einen eigenen Controller benötige, wird in der vorliegenden Arbeit nicht geteilt. Diese Trennung resultiert aus einer Aufteilung der Aufgaben zwischen View und Controller, die Letzterem alle mit Nutzereingaben verbundenen Aufgaben zuordnet. Insbesondere bei der Identifikation einer View mit einem

Aktionen übernimmt, besteht. Eine solche Komponente, im Folgenden als MVC-Komponente bezeichnet, könnte dann zur Laufzeit einer auf dem Cobamos-Framework basierenden Anwendung geladen und instanziert werden. Dadurch wäre es möglich, auf Basis einer Anwendung unterschiedliche Programmiermodelle zu unterstützen.

Wenn eine View jeweils durch ein Fenster realisiert werden soll, muss eine Möglichkeit zur gemeinsamen Verwaltung aller Views zu einem Model geschaffen werden. Das oben diskutierte MVC-Muster legt es nahe, die hierfür benötigte Funktionalität, welche auch über eine reine Fensterverwaltung hinausgehen kann[43], in den Controller zu integrieren. An dieser Stelle der Überlegungen kann dies zunächst so angenommen werden, dieser Punkt wird im Folgenden erneut aufgegriffen. Darüber hinaus sollte der bereits zuvor bei den Entwurfsüberlegungen zum Cobamos-Framework angestrebte visuelle Zusammenhalt nicht verloren gehen. Durch einzelne, von der eigentlichen Anwendung losgelöste, Fenster wäre dies jedoch wahrscheinlich. Um die Einheit der Anwendung durch die visuelle Repräsentation zu betonen, bietet sich im Bereich der Desktop-Applikationen die Verwendung so genannter MDI-Fenster (Multi Document Interface, siehe z. B. [Bal05]) an. Dadurch existieren die Fenster nicht unabhängig nebeneinander sondern werden in einem gemeinsamen „Elternfenster“ zusammengefasst. Diese Form der Umsetzung für die Sichten auf das Modell führt visuell zu einer Integration der verschiedenen Sichten in die Entwicklungsumgebung. Das Elternfenster stellt dann beispielsweise auch ein gemeinsames Menü, Werkzeugleisten und Ähnliches bereit. Diese ließen sich im Rahmen des Cobamos-Frameworks unter Verwendung des bereits diskutierten serviceorientierten Ansatzes über entsprechende Dienste von allen Kindfenstern der Anwendung – und damit von den unterschiedlichen Sichten auf das Modell – nutzen. Auch das Maximieren, Minimieren oder Schließen der Anwendung kann so für alle zugehörigen Fenster gemeinsam erfolgen.

Als Illustration des MDI-Konzeptes zeigt die unten stehende Abb. 4.11 die Anwendung Flash MX der Firma Macromedia. Diese MDI-Anwendung ist derart gestaltet, dass die einzelnen Werkzeugfenster als Kindfenster in einem gemeinsamen Elternfenster existieren. Wie es bei modernern MDI-Applikationen üblich ist, lassen sich auch in dieser Applikation die Kindfenster an verschiedenen Positionen des Elternfensters einrasten. Auf diese Art und Weise kann durch einen Nutzer eine individualisierte Arbeitsoberfläche der Anwendung erzeugt werden. Eine solche Funktionalität wird auch für das hier entworfene IDE-Framework angestrebt[44].

Fenster ist es jedoch inzwischen so, dass die grundlegenden Funktionalitäten zur Eingabe durch die View abgedeckt sind.

[43] Die Organisation eines unter Umständen notwendigen Datenaustausches zwischen den Sichten oder die Benachrichtigungen über zu behandelnde Ereignisse sind Beispiele hierfür.

[44] Die vorgenommene Realisierung dieser Funktionalität verwendet eine quelloffene und frei verfügbare Bibliothek, siehe [Wei06a].

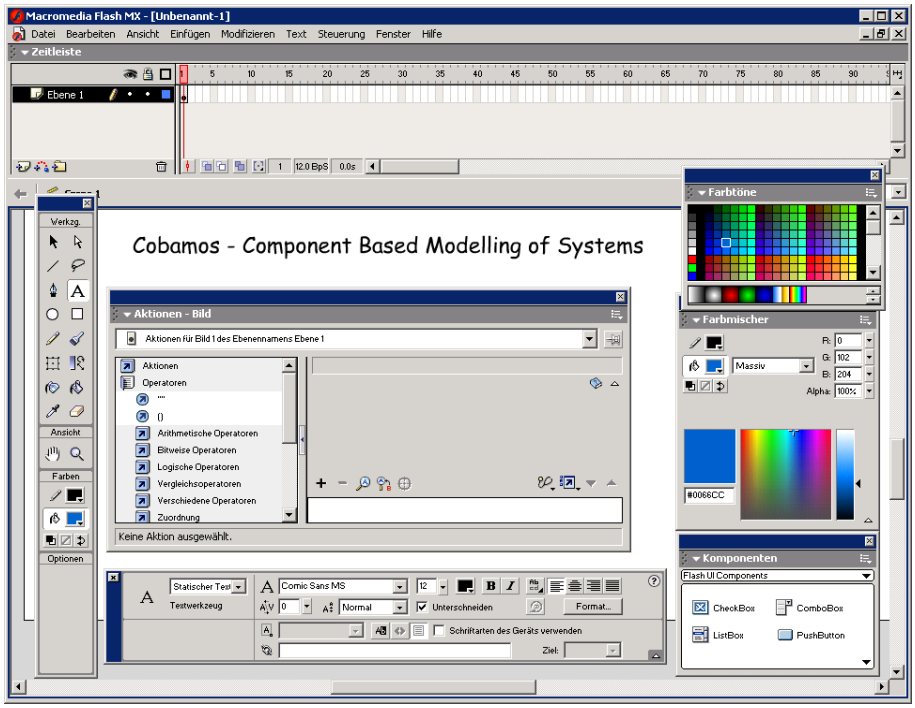


Abb. 4.11: MDI-Darstellung am Beispiel von Macromedia Flash MX

In dem hier diskutieren Anwendungsfall der Endnutzerprogrammierung sollen zu einem Programmiermodell mehrere Sichten existieren können. Diese sollen alle jederzeit den aktuellen Zustand des Modells wiedergeben. Da Änderungen über jede der Sichten[45] erfolgen können, ist eine Aktualisierung aller Sichten bei einer Änderung an den Daten des Modells notwendig. Um diese zu ermöglichen, wird das Observer-Muster (engl. Beobachter) verwendet. Die Idee dieses Verhaltensmusters besteht darin, dass sich so genannte Interessenten bei einer Ereignisquelle registrieren können und im Falle eines für sie interessanten Ereignisses benachrichtigt werden. Übertragen auf das MVC-Muster bedeutet dies, dass eine View über eine Änderung am Model informiert wird und sich daraufhin aktualisieren kann. Es sei an dieser Stelle angemerkt, dass der Einsatz dieses Musters nicht auf die Anbindung visueller Repräsentationen an sich ändernde Daten beschränkt ist.

[45] Bei der Diskussion des im Rahmen dieser Arbeit erstellten Programmiermodells wird dargelegt, dass auch Sichten sinnvoll sein können, welche einen Modellzustand – ähnlich wie ein Bericht im Falle relationaler Datenbanken – lediglich anzeigen. Da die zur Präsentation der Modelldaten verwendeten Transformationen nicht notwendig umkehrbar sind, ist es nicht immer möglich, Änderungen an solchen Darstellungen wieder auf das Modell abzubilden.

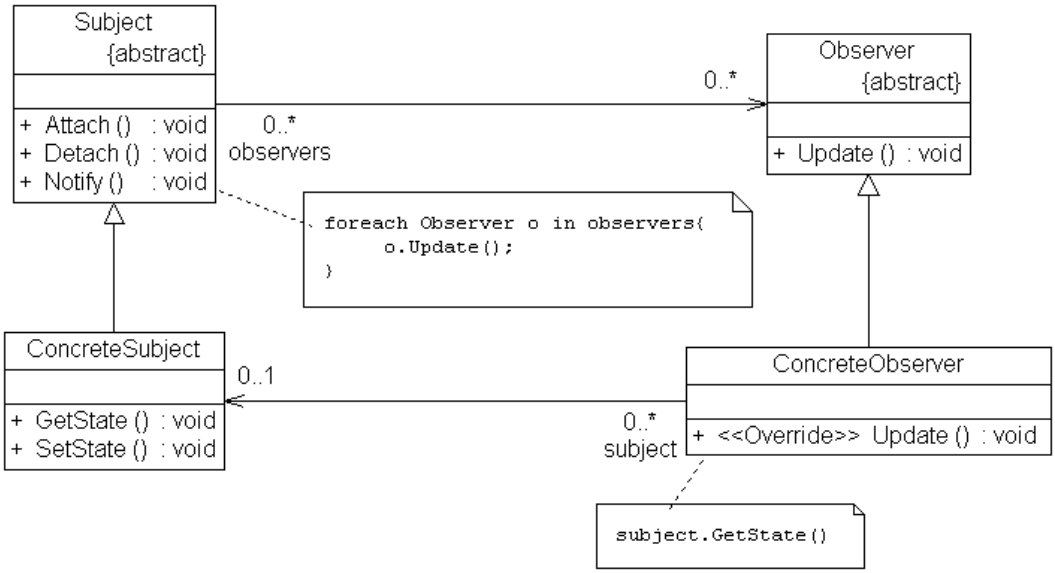


Abb. 4.12: Das Observer-Muster

In Abb. 4.12 ist das Observer-Muster in der ursprünglichen, bei [Gam95] beschriebenen, Form dargestellt. In dieser Veröffentlichung wird auch der enge Zusammenhang mit dem MVC-Muster angesprochen.

Die abstrakte Klasse `Subject` ist die Oberklasse für die konkreten zu beobachtenden Klassen und „kennt“ die Beobachter. Dabei wird kein Verweis auf einen konkreten Beobachter sondern auf die abstrakte Oberklasse `Observer` verwendet, um eine enge Kopplung zu vermeiden. Die Klasse `Subject` stellt die Methoden `Attach()` und `Detach()` bereit, mit denen Beobachter registriert bzw. entfernt werden können. Die Methode `Notify()` dient dazu, die Benachrichtigungen auszulösen[46]. Dies geschieht, indem für jede registrierte Instanz der Klasse `Observer` deren Methode `Update()` aufgerufen wird. Durch diese Methode muss die notwendige Aktualisierung des konkreten Beobachters angestoßen werden, in der Mehrzahl der Fälle wird hierfür der Zustand des konkreten beobachteten Subjekts, hier wäre dies eine Instanz der Klasse `ConcreteSubject`, abgefragt.

In Programmiersprachen, welche keine Mehrfachvererbung bieten, ist diese ursprüngliche Variante des Musters nicht optimal. Die abstrakten Oberklassen für die Subjekte und die Observer stellen in diesem Fall eine starke Einschränkung dar, da sie in eventuell bestehende Vererbungshierarchien eingefügt werden müssen. Insbesondere die abstrakte Oberklasse `Observer` stellt keine Implementation von Verhalten oder Attributen bereit, sondern dient nur der Festlegung der Schnittstelle für die Nachfahrenklassen. Dies kann jedoch auch – wenn die Programmiersprache dies zulässt – durch Verwendung eines entsprechenden Interfaces anstelle der abstrakten Klasse `Observer` erreicht werden.

Die obigen Überlegungen zeigen, dass die Verwendung des MVC-Musters im Rahmen des zu entwerfenden IDE-Frameworks es ermöglicht, dem Endnutzer unterschiedliche Sichten auf das Programmiermodell anzubieten. Sichten und Modell – die Realisierung des Controllers wird später in diesem Abschnitt behandelt – werden dabei gemeinsam in einer Komponente gekapselt und können mit denen in den vorherigen Abschnitten bereits angesprochenen Mechanismen ausgetauscht werden.

[46] Je nach gewählter Realisierung kann der Aufruf der Methode `Notify()` entweder durch das beobachtete Subjekt nach einer erfolgten Statusänderung erfolgen, oder die Verantwortung für den Aufruf liegt bei den Nutzern eines konkreten Subjektes. Dies kann sinnvoll sein, wenn mehrere Statusänderungen nacheinander durch einen Nutzer erfolgen und die entsprechenden Aktualisierungen zusammengefasst werden sollen. Zwischenzeitlich sind hierbei jedoch inkonsistente Zustände der übrigen Beobachter denkbar.

Die Verknüpfung der Sichten mit dem Modell muss durch das Framework beim Laden der Komponente vorgenommen werden. Zu diesem Zweck müssen sowohl das Modell als auch die Sichten geeignete Schnittstellen bereitstellen. Durch die Verwendung des Observer-Musters und die dadurch erreichte lose Kopplung ist es auch möglich, zu einem späteren Zeitpunkt weitere Sichten auf das Modell zu den bereits geladenen hinzuzufügen oder vorhandene zu entfernen.

Es sind sehr unterschiedliche Sichten auf das Programmiermodell denkbar. Beispielsweise kann eine der Sichten auf einer textuellen Repräsentation basieren, eine andere auf einer graphischen Darstellung. Bei einer Manipulation der Sicht durch den Nutzer muss gegebenenfalls eine entsprechende Änderung am Modell vorgenommen werden. Nach dem ursprünglichen Konzept des MVC-Musters müsste die Funktionalität für diese Modelländerungen innerhalb des Controllers bereitgestellt werden. In einem ersten Prototyp wurde dies auch derart realisiert. In diesem Zusammenhang zeigt sich jedoch ein Problem. Durch die Verschiedenartigkeit der Sichten sind die zur Abbildung der Änderungen auf das Modell notwendigen Transformationen ebenfalls verschieden. Der Controller muss also entweder eine sehr allgemeine Menge an Funktionalität zur Manipulation bereitstellen, auf die sich alle Transformationen abbilden lassen. Oder aber der Controller stellt für alle zum Entwurfszeitpunkt bekannten Sichten auf das Modell die notwendigen Abbildungen bereit. Dies bedeutet in der Folge, dass eine Ergänzung um weitere Sichten auch eine Anpassung des Controllers nach sich ziehen kann. Das Laden zusätzlicher, zum Entwurfszeitpunkt unbekannter, Sichten auf das Modell aus einer separaten Komponente ist so nicht möglich. Auch der erste Fall, dass der Controller allgemeine Funktionalitäten zum Manipulieren des Modells bereitstellt, ist fragwürdig. Die im Controller anzusiedelnden Aktionen auf dem Modell müssten, um allgemein genug zu sein, auf der Ebene atomarer Manipulationen auf dem Modell angesiedelt sein, also beispielsweise das Einfügen, Löschen oder Ändern von Entitäten im Modell umfassen. Es macht im Falle einer objektorientierten Realisierung durchaus Sinn, diese Funktionalitäten als Eigenschaften des Modells zu sehen und sie diesem zuzuordnen.

Eine View wäre in dem eben beschriebenen Szenario selbst dafür verantwortlich, die Abbildung einer Datenänderung auf das Modell bereitzustellen. Die Views müssen ohnehin die andere Richtung der Abbildung, die Transformation der Modelldaten in die entsprechende Darstellung, realisieren. Daher ist diese Zuordnung im Sinne einer hohen Kohäsion wünschenswert. Da die Transformationen spezifisch für eine Sicht sind, ist es unwahrscheinlich, dass eine Wiederverwendung dieser Transformationen über mehrere Sichten hinweg möglich ist. Sollte sich diese Möglichkeit dennoch bieten, könnten die Transformationen in Form einer gemeinsamen Oberklasse der betreffenden Sichten abstrahiert werden und in einer separaten Klassenbibliothek gekapselt werden.

Überträgt man nun noch die Aufgabe der Verwaltung der die Sichten realisierenden Fenster auf das Elternfenster und damit auf das IDE-Framework, so entfällt die Notwendigkeit einen separaten Controller einzuführen. Da die Verwaltung der Sichten bzw. Fenster durch ein gemeinsames Interface vereinheitlicht werden kann, liegt es nahe, diese im Framework anzusiedeln. Bei einer Realisierung in den austauschbaren Komponenten des Programmiermodells – beispielsweise im Controller – würde diese Funktionalität mehrfach bereitgestellt.

Nun steht die berechtigte Frage im Raum, warum das MVC-Muster überhaupt in seiner ursprünglichen Form eingeführt wurde, um es dann dahingehend zu verändern, dass dem Controller keine Aufgabe mehr zukommt. Bisher ist die Frage der Integration eines Programmiermodells und der zugehörigen Sichten in die IDE noch nicht geklärt. Das MVC-Muster in seiner ursprünglichen Form gab den entscheidenden Impuls für ein Muster, welches im Folgenden als Model-View-Connector (MVC') bezeichnet wird. Der Connector übernimmt die Aufgabe eines Integrationspunktes für das Program-

miermodell und die Sichten in die Entwicklungsumgebung und ist gleichzeitig Mediator[47] für die unterschiedlichen Sichten. Diese Form der Realisierung verringert die Anzahl der notwendigen Beziehungen zwischen den Sichten auf das Modell und der IDE. Ohne den Connector müssten sowohl das Programmiermodell als auch die einzelnen Sichten direkt mit dem IDE-Framework bzw. den zu nutzenden Diensten kommunizieren, was eine wesentlich komplexere Kommunikationsstruktur bedingen würde. In der Folge sind Anpassungen und Erweiterungen schwerer zu realisieren, die Wartbarkeit nimmt ab. Die Reduktion der Kommunikation zwischen unterschiedlichen Modulen auf eine möglichst geringe Anzahl von Schnittstellen findet sich auch in [Mey97] als eine der Regeln für modulares Design.

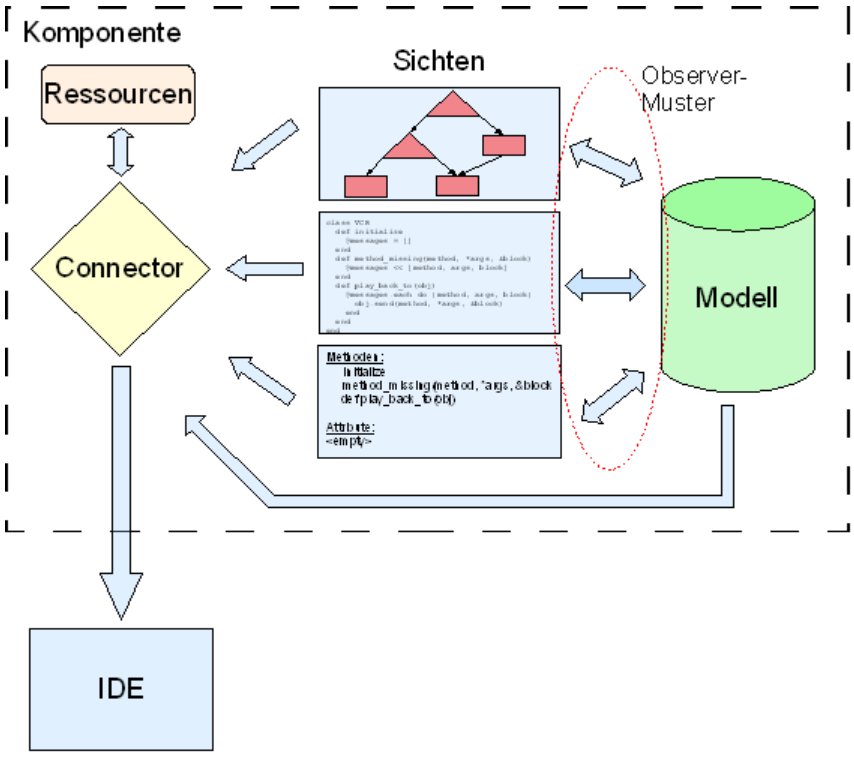


Abb. 4.13: Überblicksweise Darstellung der Kapselung von Programmiermodell und zugehörigen Sichten

In Abb. 4.13 sind die bisherigen Überlegungen zu den Möglichkeiten der Kapselung eines Programmiermodells zusammen mit verschiedenen Sichten in einer Komponente im Überblick dargestellt. Die Komponentengrenze deutet dabei eine mögliche Kapselung an; hier ist zu beachten, dass die gezeigten Referenzen vor der Instanzierung der Komponenten nicht existieren können. Innerhalb der Komponente deuten sie also die zur Laufzeit möglichen Referenzen an.

Neben der Integration in das IDE-Framework kann der Connector auch dazu verwendet werden, von den Sichten und eventuell auch dem Modell gemeinsam zu nutzende Ressourcen bereitzustellen, welche nicht durch das Framework bereitgestellt werden. Da alle Sichten und das Modell eine Referenz auf den Connector besitzen, können sie diese Daten dort abrufen. Dabei kann es sich beispielsweise um Zeichendaten oder Grafiken handeln, welche in die Komponente eingebettet sind.

[47] Das Mediator-Muster ist ein Entwurfsmuster, welches dazu dient die Kommunikation zwischen mehreren Objekten oder allgemeiner Entitäten zu vereinfachen. Zu diesem Zweck wird ein zentraler Vermittler, der Mediator, in das Design eingeführt.

Zudem wird in der Darstellung nochmals deutlich, dass die Sichten Referenzen sowohl auf den Connector, als auch auf das Modell[48] besitzen sollen. Dies muss sich in der entsprechenden Schnittstelle – diese wird als `IView` bezeichnet – niederschlagen. Darüber hinaus muss eine Sicht das Observer-Muster unterstützen. Da dessen Realisierung im Grunde unabhängig von den Sichten ist, wird hierfür eine getrennte Schnittstelle eingeführt. Dies ermöglicht es später, bei Bedarf Entitäten zu realisieren, die zwar auf Modelländerungen reagieren können, aber keine Sicht im Sinne der obigen Überlegungen darstellen.

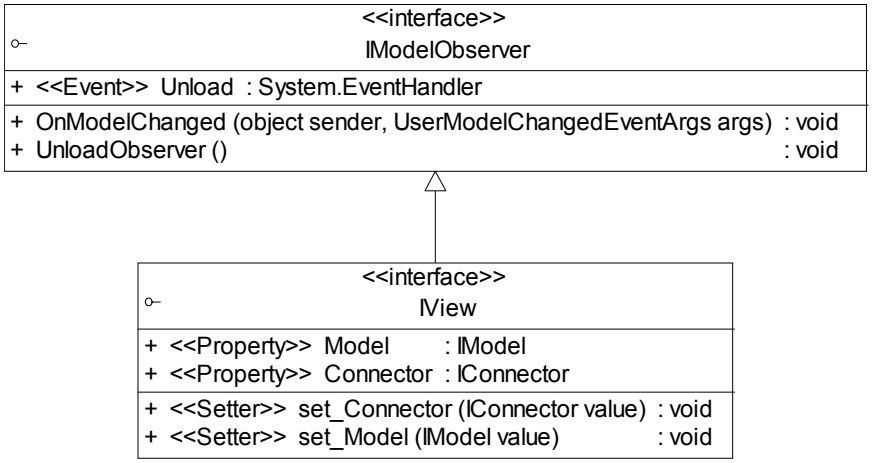


Abb. 4.14: Die Schnittstellen IModelObserver und IView[49]

In Abb. 4.14 ist die Schnittstelle `IModelObserver` für die reinen „Modellbeobachter“ und davon abgeleitet die Schnittstelle `IView` für die Sichten dargestellt. Die gezeigten Schnittstellen entstammen der vorgenommenen prototypischen Realisierung. Die Referenzen der Schnittstelle `IView` auf das Modell und auf den Connector können in der gezeigten Variante nur gesetzt werden, ein Lesezugriff auf das Modell oder den Connector soll über die Schnittstelle `IView` nicht möglich sein.

Die Methode `OnModelChanged()` entspricht der Methode `Update()` aus der Darstellung des Observer-Musters in Abb. 4.12 und wird vom Modell im Falle einer relevanten Änderung aufgerufen. Die veränderte Signatur der Methode rührt daher, dass das Observer-Muster in der prototypischen Realisierung unter Verwendung von Ereignissen (im Diagramm gekennzeichnet durch das Stereotyp `<<Event>>`) umgesetzt wird. Dies ermöglicht lediglich eine etwas elegantere Implementation. Die benötigte Funktionalität lässt sich – so wurde das Observer-Muster ja auch eingeführt – durch die Verwendung von Referenzen auf die Instanz, zu der die aufzurufende Methode gehört, in analoger

[48] Diese Situation spiegelt den bisherigen Stand der Überlegungen wieder. Es ist keine zwingende Referenz vom Connector zu den Sichten erforderlich. Aufgrund zusätzlicher Anforderungen können diese jedoch notwendig werden. Ein solches Szenario wird in Kapitel 8 angesprochen. Hier tritt die Situation auf, dass einige Sichten untereinander kommunizieren müssen. Um eine enge Bindung der Sichten untereinander zu vermeiden, wird auch für diese Kommunikation der Connector als Mediator verwendet und besitzt in der Folge Referenzen zu einigen der Sichten.

[49] Um anzuzeigen, dass auf die Eigenschaften (gekennzeichnet durch das Stereotyp `<<Property>>`) `Model` und `Connector` der Schnittstelle `IView` nur ein Schreibzugriff möglich ist, erzeugt das verwendete Modellierungswerkzeug Powerdesigner eine Methode zum Setzen der Werte (gekennzeichnet durch das Stereotyp `<<Setter>>`), aber keine Methoden für einen lesenden Zugriff. Im Falle einer Realisierung in C# müssen diese Methoden nicht gesondert implementiert werden, die entsprechenden Beschränkungen des Zugriffs können durch die Deklaration der Eigenschaften realisiert werden.

Weise realisieren. Die Darstellungen im Folgenden erfolgen auf einer Entwurfsebene, auf der dieses Umsetzungsdetail keine wesentliche Rolle spielt. Eine geeignete Realisierung kann bei der Umsetzung gewählt werden.

Diese Realisierung unter Verwendung von Ereignissen schlägt sich auch in der Schnittstelle `IModel`, dargestellt in Abb. 4.15, nieder. In dieser existiert das Ereignis `Changed`, etwas vereinfacht kann man sagen, dass dies einer Liste mit Referenzen auf die bei einer Benachrichtigung der Beobachter aufzurufenden Methoden entspricht.

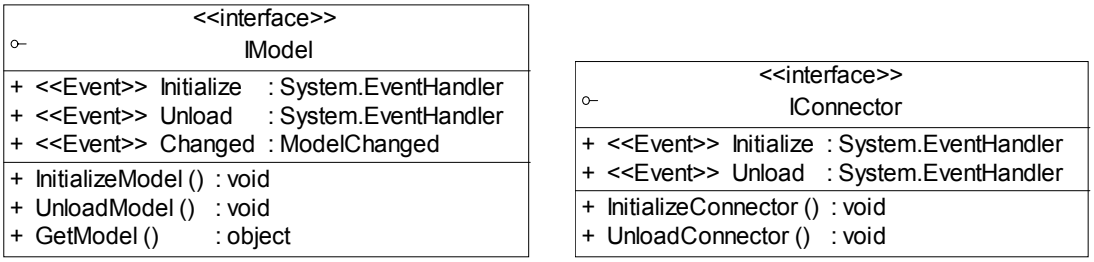


Abb. 4.15: Die Schnittstellen IModel und IConnector

Neben den durch die Überlegungen zum MVC'- bzw. Observer-Muster motivierten Methoden, Eigenschaften und Ereignissen definieren die Schnittstellen Methoden und Ereignisse zum Initialisieren bzw. Entladen. Das Teilsystem des Frameworks, welches das Modell, die Sichten und den Connector verwaltet, wird im Folgenden als MVC'-Verwaltung bezeichnet. Dieses Teilsystem ruft beim Laden der jeweiligen Entitäten die eben angesprochenen Methoden auf, damit diese die Möglichkeit haben, nach dem Laden notwendige Initialisierungen bzw. vor dem Entladen nötige Aufräumarbeiten durchzuführen. Die Methoden lösen, bevor sie wieder verlassen werden, das zugehörige Ereignis aus. Auf die Notwendigkeit dieser Ereignisse wird weiter unten genauer eingegangen.

Modell, Connector und Sichten müssen nicht gemeinsam in einer Komponente gekapselt sein, sie können beispielsweise auch jeweils in einer eigenen Komponente bereitgestellt werden. Es ist jedoch sinnvoll, zumindest das Modell, den Connector und die unbedingt zur Arbeit mit dem Modell notwendigen Sichten in einer Komponente zusammenzufassen, da diese dann mit einem einzigen Ladevorgang instanziert und bereitgestellt werden können.

Im Folgenden soll die für das Laden im IDE-Framework benötigte Funktionalität genauer dargestellt werden. Die Möglichkeit, Komponenten zur Laufzeit einer Anwendung dynamisch zu laden, wurde bereits in Abschnitt 4.3.2 angesprochen. Die Reihenfolge, in der die einzelnen Entitäten – sei es aus einer oder mehreren Komponenten – geladen und instanziert werden, ist nicht beliebig. Da das Modell zum Funktionieren keine Referenzen auf die Sichten benötigt, diese aber ohne das Modell nicht sinnvoll sind, muss dieses zuerst geladen und instanziert werden. Da es sich beim Connector um den Verbindungspunkt zum IDE-Framework handelt und er auch einen eventuellen Zugriff auf zusätzliche gemeinsame Ressourcen bereitstellen soll, ergibt sich nahe liegender Weise die Reihenfolge Connector, Modell, Sichten. Dann können beispielsweise bei der Initialisierung auftretende Fehler schon unter Verwendung der bereitgestellten Frameworkfunktionalität behandelt werden.

Da nur ein Modell aktiv sein soll – dies ist eine Entwurfsentscheidung, die Ansätze ließen sich analog auf die gleichzeitige Verwaltung mehrer Modelle mit den zugehörigen Sichten erweitern – wird beim Laden eines Modells ein eventuell vorhandenes entladen. Dabei werden auch alle zu diesem Zeitpunkt geladenen Sichten entfernt, da diese in der Regel nur mit dem zugehörigen Modell zusammenarbeiten

können. Ein alleiniges Austauschen des Modells unter Beibehaltung der geladenen Sichten würde einen geeigneten Mechanismus erfordern, um die Verträglichkeit mit den Sichten sicherzustellen. Denkbar wäre eine Ergänzung der beteiligten Entitäten um Metainformationen in Attributen oder aber auch das Einführen von spezielleren Interfaces, die von den hier vorgestellten allgemeinen abgeleitet sind. Diese Möglichkeiten wurden nicht weiter verfolgt, da der Aufwand für den zu erwartenden Nutzen im Hinblick auf die Zielsetzung der Arbeit nicht gerechtfertigt erschien.

Das in Abb. 4.16 gezeigte Sequenzdiagramm gibt einen Überblick über die beim Laden von Modell, Connector und Sichten notwendigen Abläufe. Dabei ist der für die Behandlung der Sicht zuständige Teil des Ablaufes – beginnend bei Nachricht 11 – ggf. für alle zu ladenden Sichten zu wiederholen.

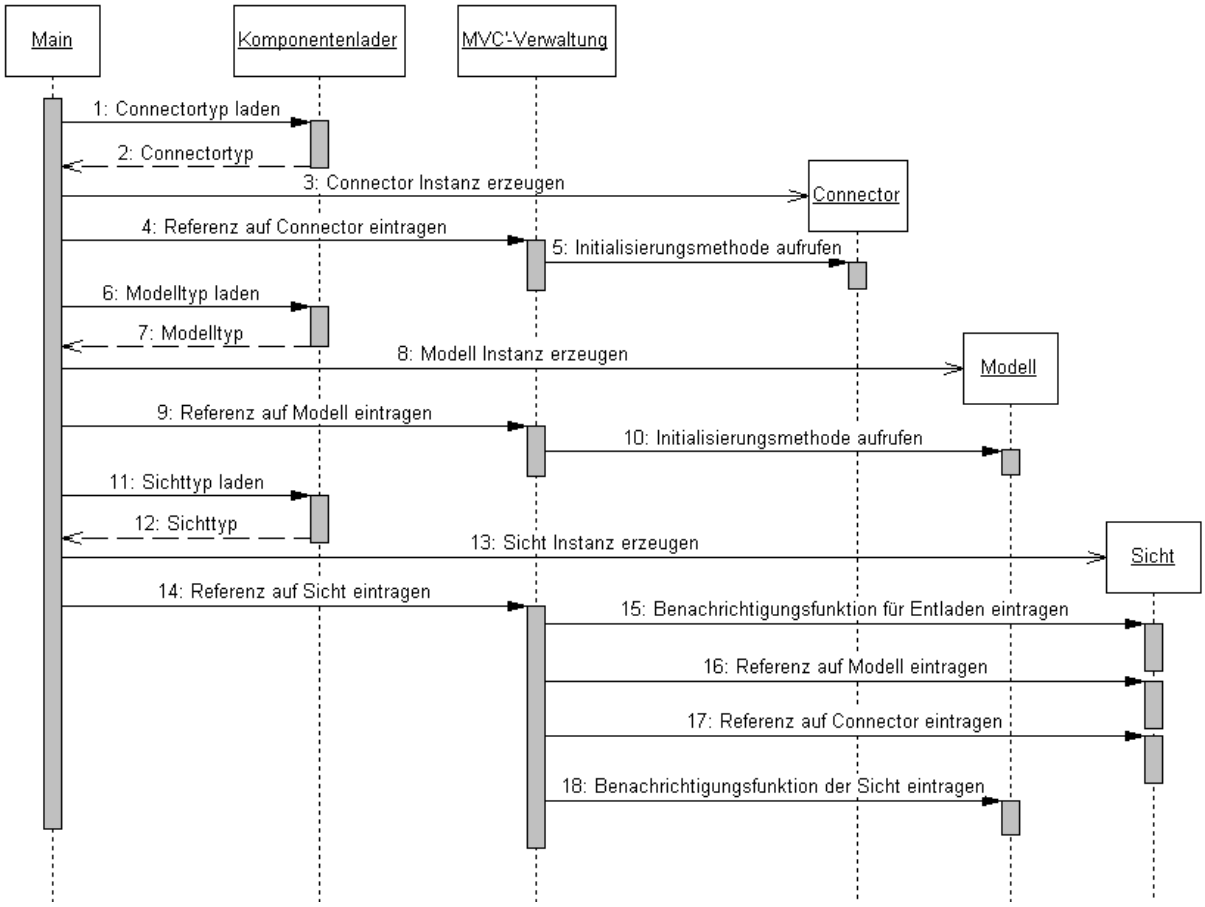


Abb. 4.16: Laden von Modell, Connector und Sicht

In dieser Realisierung wird das Laden der einzelnen Typen durch das Hauptprogramm bzw. durch einen Nutzer über das Hauptfenster angestoßen. Denkbar wäre auch eine Verlagerung dieser Funktionalität in die MVC'-Verwaltung, hieraus ergeben sich keine besonderen Vor- oder Nachteile. Die bereits angesprochen Initialisierungsmethoden werden von der MVC'-Verwaltung aufgerufen (siehe Nachrichten 5 und 10), wenn eine entsprechende Entität hinzugefügt wird. Wenn eine Sicht zur MVC'-Verwaltung hinzugefügt wird, so wird diese zunächst konfiguriert. Dabei bekommt die Sicht zunächst eine Benachrichtigungsfunktion übergeben (Nachricht 15). Diese muss von der Sicht aufgerufen werden, wenn sie entladen werden soll, und dient der Benachrichtigung der MVC'-Verwaltung. Anschließend werden die Referenzen auf das Modell und den Connector an die Sicht übertragen (Nachrichten 16 und 17). Nachdem die Sicht konfiguriert ist, wird ihre Benachrichtigungsfunktion für Modelländerungen beim Modell eingetragen.

Soll eine einzelne Sicht später zu den bereits vorhandenen hinzugefügt werden, kann dies analog zu der bereits dargestellten Funktionalität erfolgen. Der hierfür relevante Teil der in Abb. 4.16 dargestellten Funktionalität (Nachrichten 11 bis 18) ist unabhängig von den vorherigen Abläufen, solange vorausgesetzt werden kann, dass Modell und Connector bereits geladen und instanziert sind.

Beim Entladen muss darauf geachtet werden, das Modell nicht vor den Sichten zu entfernen. Anderenfalls wären bei der Implementation entsprechende Maßnahmen umzusetzen, um Fehler durch ins Leere zeigende Referenzen abzufangen. Zusätzlich müsste verhindert werden, dass über die Sichten noch eine Interaktion nach dem Entladen des Modells möglich ist. Da der Connector wie oben bereits angesprochen sowohl für das Modell als auch für die Sichten den Zugriff auf die Frameworkfunktionalität und zusätzliche gemeinsame Ressourcen ermöglicht, muss dieser bis zuletzt verfügbar sein.

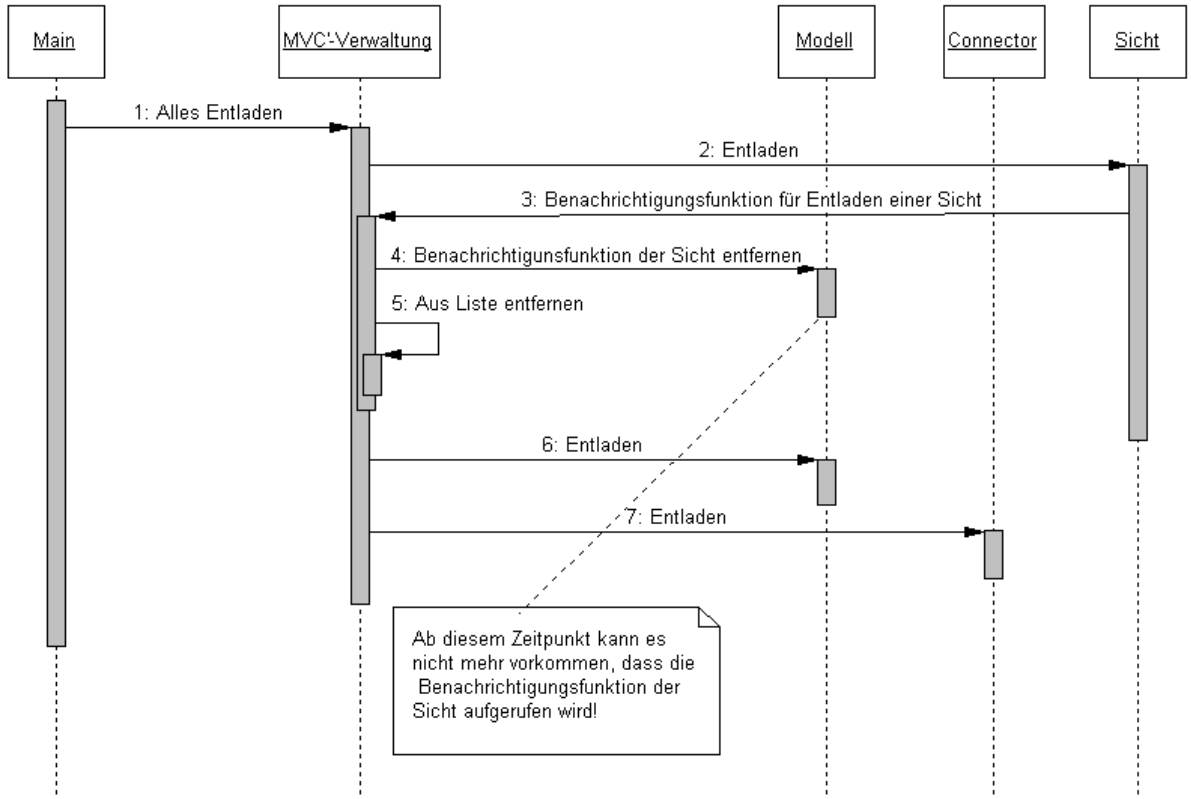


Abb. 4.17: Entladen von Modell, Connector und Sicht

Das in Abb. 4.17 präsentierte Sequenzdiagramm verdeutlicht den beim Entladen aller Entitäten notwendigen Ablauf. Das Entladen der Sicht erscheint auf den ersten Blick unnötig kompliziert, da der Aufruf zum Entfernen der über die Benachrichtigungsfunktion bestehenden Verbindung zum Modell (realisiert durch Nachricht 4) nicht von der Sicht selbst ausgeht. Vielmehr wird dies durch den Aufruf einer Benachrichtigungsfunktion (dies geschieht mit Nachricht 3) an die MVC'-Verwaltung delegiert. Dies geschieht aus zwei Gründen. Zum einen muss die Sicht beim Entladen auch aus der Liste der Sichten in der MVC'-Verwaltung entfernt werden. Dies ist in diesem Szenario, bei dem das Entladen der Sicht von der MVC'-Verwaltung angestoßen wird, auch anders realisierbar. Es kann jedoch auch der Fall eintreten, dass das Entladen einer Sicht von dieser selbst initiiert wird, beispielsweise im Falle eines schwerwiegenden Fehlers. Da die Sicht selbst keinen Zugriff auf die MVC'-Verwaltung besitzt, kann die existierende Referenz nicht direkt entfernt werden. Durch die Realisierung über die

Benachrichtigungsfunktion kann dieses Entfernen, unabhängig vom Auslöser, in allen Fällen gleich erfolgen. Zum anderen verfügt zwar eine Sicht, welche die Schnittstelle `IView` (siehe Abb. 4.14) implementiert, über eine Referenz auf das Modell und könnte das Entfernen der eigenen Benachrichtigungsfunktion selbst durchführen. Es sollen jedoch auch einfach „Modellbeobachter“ möglich sein. Diese implementieren nur die Schnittstelle `IModelObserver` (siehe ebenfalls Abb. 4.14), in dieser existiert keine Referenz auf das Modell. Das gezeigte Vorgehen ermöglicht es, Implementationen dieser Schnittstelle ebenfalls im Rahmen der MVC'-Verwaltung zu verwenden. Dabei kann die Behandlung in beiden Fällen vollkommen analog erfolgen. Um den angesprochen Ansatz realisieren zu können, existiert in der Schnittstelle `IModelObserver` das Ereignis `Unload`.

In den Schnittstellen für das Modell und den Connector (siehe Abb. 4.15) sind analog zu den obigen Überlegungen ebenfalls Ereignisse deklariert, welche bei der Initialisierung oder dem Entladen aufgerufen werden. Diese sind derzeit als Basis für zukünftige Erweiterungen vorgesehen.

Auch die Funktionalität zur Integration der Sichten in die IDE wird von der MVC'-Verwaltung bereitgestellt. Sie führt eine Liste aller geladenen Views und besitzt eine Referenz auf Model und Connector. Um dem Nutzer der Anwendung die Möglichkeit zu geben, bereits vorhandene Views zu nutzen, weitere hinzuzufügen oder auch vorhandene zu entladen, ist eine Möglichkeit zur Interaktion nötig. Diese muss dem Nutzer auch ermöglichen, auf Sichten, die aktuell verborgen[50] sind, zuzugreifen. Da die Oberflächenfunktionalität unabhängig von der eigentlichen MVC'-Verwaltung ist, sollte diese getrennt realisiert werden, um einen unter den Gesichtspunkten des Software Engineerings guten Entwurf zu erhalten. Da diese Oberfläche jederzeit den aktuellen Zustand hinsichtlich der geladenen Views anzeigen soll, kann hier wieder das Observer-Muster Anwendung finden. Die Nutzungsoberfläche wird als Observer der MVC'-Verwaltung implementiert, diese löst bei jeder Änderung (neues Modell geladen, Sicht entfernt o. Ä.) eine Benachrichtigung aus und ermöglicht der Nutzungsoberfläche damit eine Aktualisierung. Damit die Nutzungsoberfläche Zugriff auf die benötigten Daten der MVC'-Verwaltung erhält, müssen in dieser entsprechende Zugriffsmethoden vorgesehen werden.

[50] Verborgen bedeutet in diesem Zusammenhang „nicht sichtbar“, die Sichten sind jedoch instanziert. Da die Übersicht nur die Sichten anzeigt, auf die die MVC'-Verwaltung eine Referenz besitzt, können keine nichtinstanzierten Sichten in der Liste auftreten. Prinzipiell ist eine Realisierung denkbar, in der die Sichten nicht beim Laden der Komponente, sondern bei der ersten Anforderung erzeugt werden. Dies würde lediglich eine zusätzliche Verwaltung von in Komponenten verfügbaren, aber noch nicht instanzierten Sichten erfordern. Die prototypische Realisierung instanziert beim Laden einer entsprechenden Komponente alle in dieser verfügbaren Views. Da die Möglichkeit besteht, Sichten zur Laufzeit laden und entladen zu können, kann durch die Aufteilung von Sichten auf einzelne Komponenten jedoch ein zum Fall des „Instanzierens bei Bedarf“ ähnliches Verhalten erreicht werden.

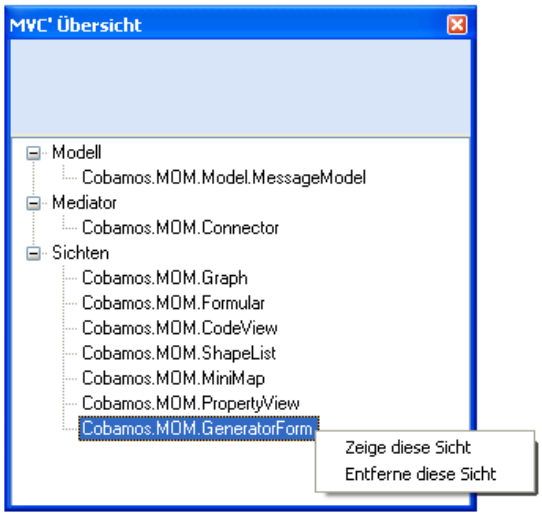


Abb. 4.18: Exemplarische Realisierung einer Übersicht über die geladenen Sichten

In der prototypischen Realisierung ist die Benutzungsoberfläche für die MVC'-Verwaltung in Form eines Fensters realisiert, welches die notwendigen Informationen in einer Baumansicht darstellt. Dieses ist in Abb. 4.18 dargestellt; die angezeigten Daten beziehen sich auf das im weiteren Verlauf dieser Arbeit auf Basis des Cobamos-Frameworks umgesetzte Programmiermodell (zur Realisierung siehe Kapitel 8, die konzeptuellen Grundlagen werden in Kapitel 7 diskutiert). Die Bezeichner der angezeigten Entitäten sind in diesem Beispiel noch die Bezeichner der sie realisierenden Klassen. Diese während der Entwicklung des Systems nützliche Benennung sollte bei der Anwendung mit Endnutzern geändert werden. Dem Nutzer steht zur Interaktion in dieser ersten Realisierung ein Kontextmenü zur Verfügung, dieses bietet abhängig vom gewählten Element unterschiedliche Interaktionsmöglichkeiten an. Im Beispiel wurde eine Sicht ausgewählt, der Benutzer kann nun entscheiden, ob er diese anzeigen oder entladen möchte. Letzteres bezieht sich auf das vollkommene Entfernen der Sicht, um sie erneut anzeigen zu können, müsste die Komponente, in der sie enthalten ist, erneut geladen werden.

4.4.3 Ergänzungen

Die im vorherigen Abschnitt 4.4.2 dargestellte Methode zur Anbindung unterschiedlicher Sichten an ein Programmiermodell und zur Kapselung von Sichten und Modell in einer gemeinsamen Komponente erlaubt eine Erweiterung, um einige weitere im Rahmen des Einsatzes in der Endnutzerprogrammierung wichtige Punkte zu adressieren.

Die Qualität der Fehlermeldungen der verwendeten Werkzeuge – diese Überlegungen beziehen sich nicht auf Applikationsfehler der Werkzeuge sondern auf Benutzerfehler – kann einem Programmierer die Realisierung einer Softwareapplikation erheblich erleichtern oder sie aber auch – durch irreführende Fehlermeldungen – erschweren. Für einen Endnutzer mit nur geringen Kenntnissen in der Softwareentwicklung ist neben dem Inhalt der Fehlermeldung auch der Zeitpunkt der Rückmeldung von Bedeutung. Je unmittelbarer ein entsprechendes Feedback auf eine fehlerhafte oder fragwürdige Aktion folgt, desto leichter ist es, einen Fehler zu lokalisieren und zu beheben. Dies gilt zwar auch für Programmierer, aufgrund des höheren Kenntnisstandes sind diese aber im Allgemeinen in der Lage, auch aufgrund von späteren Rückmeldungen – beispielsweise während des Übersetzungsvorgangs – Entsprechendes zu leisten.

Da alle Manipulationen am Programmiermodell über die jeweiligen Sichten vorgenommen werden, gibt es prinzipiell zwei Stellen, an denen eine Fehlerprüfung und das Auslösen eines eventuell notwendigen Feedbacks für den Anwender geschehen können. Die Sicht, über die das Modell manipuliert wird, kann diese Aufgabe übernehmen. Im Sinne einer möglichst weiten Wiederverwendung der Funktonalität zur Fehlerprüfung sollte sich diese aber nur auf die für diese Sicht spezifischen Fehler beschränken. Dies wäre beispielsweise in einer textuellen Darstellung das Fehlen eines Zeichens zur Markierung des Zeilenendes. Fehler, die auf der Ebene des Programmiermodells anzusiedeln sind, wie etwa eine Verknüpfung zweier nicht zusammenpassender Entitäten, sollten durch das Modell geprüft werden. Dies geschieht sinnvollerweise vor dem Ausführen der Aktionen, die das Modell zur Manipulation seines Zustandes zur Verfügung stellt. Um das Beispiel wieder aufzugreifen, wenn das Modell eine Methode zum Verbinden zweier Entitäten bietet, müsste diese vor einer Änderung am Modellzustand die Zulässigkeit dieser Aktion prüfen und gegebenenfalls den Anwender entsprechend benachrichtigen.

Ein weiterer bisher noch nicht betrachteter Punkt ist die Ausführbarkeit der modellierten Anwendung. Hier sind grundsätzlich zwei Varianten denkbar. Bei der einen ist in die Komponente, welche das Modell realisiert, bereits die für eine Simulation benötigte Funktionalität integriert. Dies wäre beispielsweise möglich, wenn es sich um ein Level-Raten-Modell, wie es in einigen Arbeiten im Bereich der Sportinformatik Verwendung findet [Per02], handelt. Ein Ausführen der Anwendung würde dann dem Durchführen einer Simulation entsprechen. In diesem Fall kann durch eine geeignete Sicht auf das Modell der sich ändernde Zustand desselben dargestellt werden.

Die zweite Variante besteht darin, dass das Modell in eine ausführbare Form – dabei handelt es sich im Fall der Endnutzerprogrammierung um die Zielanwendung – überführt werden muss. Hier werden noch keine Annahmen über die Realisierung dieses Schrittes, ob dies beispielsweise durch einen Compilationsvorgang oder einen interpreterartigen Ansatz geschieht, gemacht. Interessant ist für die Betrachtungen an diesem Punkt die Frage, wie die notwendigen Funktionalitäten an das Modell angebunden werden können.

Es ist denkbar, die Transformation des Modells in eine ausführbare Form als analog zu den Transformationen in eine andere Präsentation, wie sie bei der Anbindung der Sichten an das Modell stattfindet, zu betrachten. Dann ist es jedoch nahe liegend, auch diesen Transformationsschritt wie eine der anderen Sichten anzubinden. Dies hat neben der Möglichkeit der Gleichbehandlung bei der internen Verwaltung der Transformationen bzw. Sichten auch den Vorteil, dass die „Ausführung“ des Modells wie alle anderen Sichten über die Änderungen am Modellzustand benachrichtigt wird. Auf dieser Basis können Anwendungen erstellt werden, bei denen sich eine Änderung am Modell direkt auf dessen Ausführung auswirkt.

4.5 Zusammenfassung

In diesem Kapitel wurden die zentralen Aspekte des Entwurfs für das Cobamos-Framework vorgestellt. Die serviceorientierte Architektur, die im Cobamos-Framework als Basis zur Bereitstellung von Funktionalität dient, adressiert die am Beginn des Kapitels (siehe Abschnitt 4.2.2) aufgestellte Anforderung nach möglichst umfangreicher Wiederverwendung der Framework-Funktionalität. Da wiederzuverwendende Funktionalität in den Diensten gekapselt ist, kann explizit festgelegt werden, welche Teile des Frameworks zur Wiederverwendung gedacht sind. Durch die Verwendung von Komponenten bei der Realisierung der Dienste wird für diese gemeinsame Funktionalität die Forderung nach Anpassbarkeit (gefordert in Abschnitt 4.2.3) erfüllt. Auch Erweiterungen (dies wurde in Abschnitt 4.2.4 gefordert) der bereitgestellten Funktionalität sind durch

das Hinzufügen neuer Komponenten möglich, wenn diese die entsprechenden Schnittstellen für Dienste im Cobamos-Framework implementieren.

Der Anforderung nach einfacher Nutzbarkeit (siehe Abschnitt 4.2.5) einer unter Verwendung des Frameworks realisierten Anwendung wird durch die Bereitstellung von Diensten sowohl zur Kommunikation mit dem Anwender als auch zur Integration in die gemeinsame Oberfläche der Entwicklungsumgebung angesprochen. Die Nutzbarkeit des Frameworks durch Softwareentwickler wurde beim Entwurf der zu realisierenden Schnittstellen und Anpassungsmechanismen berücksichtigt und hat die entsprechenden Entwurfsentscheidungen beeinflusst. Die Erfüllung dieser Anforderung lässt sich ohne die Durchführung einer Nutzungsstudie nicht verlässlich beantworten, dies ist im Rahmen dieser Arbeit jedoch nicht zu leisten.

Die Dienste stellen für alle anderen Bestandteile des Frameworks Funktionalität bereit, insbesondere auch zur Nutzung durch die Kommandokomponenten und die unter Verwendung des MVC'-Musters gekapselten Programmiermodelle. Das Konzept des MVC'-Musters ermöglicht die Kapselung eines Programmiermodells und der zugehörigen Sichten in Komponenten (eine getrennte Kapselung ist aber nicht erforderlich), durch diesen Ansatz wird die Anforderung nach der Unterstützung unterschiedlicher Programmiermodelle (vgl. Abschnitt 4.2.1) erfüllt. Über den Connector können sowohl das Modell als auch die Sichten auf die Dienste des Frameworks zugreifen und diese Funktionalität nutzen. Die für den Austausch eines Programmiermodells notwendige Interaktion eines Endanwenders entspricht dem Laden einer anderen Datei – der Komponente, die das Modell kapselt.

Um auch die durch den menschlichen Nutzer einer Anwendung über die Interaktionsoberfläche ansprechbare Funktionalität erweitern zu können, wurde in diesem Kapitel das Konzept der Kommandokomponenten vorgestellt. Diese stellen eine abgeschlossene Funktionalität bereit und werden in die Nutzungsoberfläche der Anwendung integriert. Durch die Kapselung in Komponenten kann auch hier vorhandene Funktionalität durch das Austauschen der entsprechenden Komponenten angepasst werden; eine Erweiterung ist durch das Hinzufügen zusätzlicher Komponenten, welche die entsprechenden Schnittstellen implementieren, möglich. Die zum Austausch bzw. Hinzufügen von Komponenten in einer auf dem Framework basierenden Anwendung notwendigen Schritte beschränken sich auf das Kopieren von Dateien in ein Verzeichnis. Diese Realisierung erfüllt die optionale Anforderung, dass Anpassungen und Erweiterungen durch Endnutzer möglich sein sollen.

Die Beschreibung des Entwurfs für das Cobamos-Framework erfolgte soweit als möglich unabhängig von einer konkreten Realisierung. Wie in der Einleitung angesprochen, ist die Umsetzbarkeit der in dieser Dissertation erarbeiteten Konzepte ein wichtiger Aspekt. In Kapitel 6 wird daher auf einige interessante Details der Implementation eingegangen. Zuvor wird jedoch im folgenden Kapitel die Wahl der Implementationssprache begründet.

5 Das .NET-Framework als Komponentenmodell

„Well, now we have a ton of .NET vaporware that even Microsoft does not expect to ship for years.“
(Nun haben wir eine Tonne .NET-Luftsoftware, von der nicht einmal Microsoft erwartet, dass sie sie in den nächsten Jahren ausliefern.)
Robert Metcalfe, Ethernet-Miterfinder und Gründer der Firma 3Com, 2000

Da die im Verlauf dieser Arbeit vorgestellten Überlegungen und Entwürfe auch – zumindest in prototypischen Realisierungen – umgesetzt werden sollen, muss eine Festlegung auf eine Programmiersprache erfolgen. In diesem Kapitel wird die Wahl der Implementationssprache C# begründet. Im weiteren Verlauf dieses Kapitels wird das .NET-Framework der Firma Microsoft in seinen Kernpunkten vorgestellt, welches – anders als es Robert Metcalfe in seiner Kolumne in der Ausgabe vom 30. Juni 2000 der Zeitschrift InfoWorld vermutete – in der Zwischenzeit bereits in der Version 2.0 verfügbar ist, der Nachfolger befindet sich bereits in der Entwicklung. Die Betrachtung konzentriert sich insbesondere auf die Eigenschaften des Frameworks, die im Hinblick auf eine Verwendung als Komponentenframework maßgeblich sind. Neben dem grundsätzlichen Aufbau der Laufzeitumgebung werden die Sprachinteroperabilität des .NET-Frameworks und das Assemblykonzept behandelt. Letzteres wird auch im Hinblick auf den Zusammenhang mit der Komponentendefinition aus Kapitel 3 besprochen.

5.1 Einleitung und Wahl einer Implementationssprache

Nachdem der grundlegende Entwurf des IDE-Frameworks in den vorangegangenen Kapiteln festgelegt wurde, muss eine Programmiersprache für die Implementation gewählt werden. Diese Sprache muss eine Umsetzung des Entwurfs ermöglichen und die aus diesem resultierenden Anforderungen an den Sprachumfang erfüllen.

Da der Entwurf in weiten Teilen auf der Verwendung von Komponenten beruht, und diese dynamisch, d. h. zur Laufzeit einer Anwendung geladen und instanziert werden sollen, ist eine Programmiersprache zu wählen, die solche Möglichkeiten bietet. Es ist wünschenswert, beim Entwurf der einzelnen Komponenten die vielfältigen und sehr flexiblen Möglichkeiten der Objektorientierung zur Verfügung zu haben, daher besitzt die Implementationssprache idealerweise sowohl komponenten- als auch objektorientierte Züge.

Zum Zeitpunkt der anfänglichen Überlegungen zu dieser Arbeit stellte Microsoft die erste Version des .NET-Frameworks der Öffentlichkeit zur Verfügung. In diesem Zuge wurde auch die Programmiersprache C# veröffentlicht (siehe z. B. [ECM05a] für die Sprachspezifikation und [Mic06c] für Microsofts Produkte rund um diese Programmiersprache). Es handelt sich dabei um eine objektorientierte Programmiersprache mit einer C-ähnlichen Syntax und sehr weitgehender Unterstützung für komponentenbasierte Softwareentwicklung. Die gängigen imperativen und objektorientierten Sprachkonstrukte sind ebenfalls in dieser Programmiersprache vorhanden. C# unterstützt keine Mehrfachvererbung, wie die Programmiersprache Java bietet die Sprache stattdessen die Möglichkeit zur Definition und Nutzung von Schnittstellen. Mit den so genannten Delegaten existiert in C# ein rein objektorientierter und typsicherer Ansatz zur Realisierung von Funktionszeigern. Letzteres ist bei der Realisierung der auf dem Observer-Muster basierenden Teile des Cobamos-Frameworks von Interesse.

Da sowohl diese neue Sprache als auch die Technologie des zugehörigen .NET-Frameworks viele interessante Konzepte umsetzen – das Versionierungskonzept ist ein Beispiel hierfür, siehe Abschnitt

5.3 Assemblys – lag es nahe, diese Arbeit und die zu erstellende prototypische Implementation gleichzeitig dafür zu nutzen, Erfahrungen mit der Entwicklung von Applikationen auf dieser Basis zu sammeln.

In Kapitel 3 wurde der Begriff der Softwarekomponente auf der konzeptionellen Ebene definiert. Bei der praktischen Anwendung des Komponentenkonzeptes im Rahmen der Anwendungsentwicklung ergeben sich weitergehende Anforderungen. Außerdem ist zum Einsatz von Komponenten immer eine Ausführungsumgebung notwendig. In den folgenden Abschnitten dieses Kapitels wird das .NET-Framework hinsichtlich seiner für die komponentenbasierte Softwareentwicklung relevanten Eigenschaften beschrieben. Detailliertere Informationen zu Aspekten der Implementation des Cobamos-Frameworks mit der Programmiersprache C# finden sich in Kapitel 6.

5.2 CLR und CLI – Aufbau des .NET-Frameworks

In diesem Abschnitt soll die grundlegende Struktur des .NET-Frameworks dargestellt werden. Die Betrachtung erfolgt auf einer sehr hohen Abstraktionsebene und dient hauptsächlich dazu, Begrifflichkeiten zu klären und zentrale Sachverhalte zu erläutern.

Die **Common Language Runtime** (CLR, Gemeinsame Sprach-Laufzeitumgebung) ist der zentrale Bestandteil des .NET-Frameworks. Es handelt sich dabei um Microsofts kommerzielle Implementation zum Standard **Common Language Infrastructure** (CLI, Gemeinsame Sprachinfrastruktur). Bei der CLI handelt es sich um einen von der European Computer Manufacturers Association (ECMA)[51] inzwischen in der dritten Version [ECM05b] verabschiedeten Standard. Dieser wurde mit dem Ziel entworfen, die Interoperabilität zwischen Programmiersprachen zu ermöglichen. Zu diesem Zweck definiert der CLI-Standard das **Common Type System** (CTS, Gemeinsames Typsystem), die zugehörigen **Metadaten**, die **Common Language Specification** (CLS, Gemeinsame Sprachspezifikation) und das **Virtual Execution System** (VES, Virtuelles Ausführungsumgebung). Beim VES handelt es sich um die Ausführungsumgebung für CLI-kompatible Programme. Das CTS beschreibt das Typsystem für die CLI, also eine Menge von grundlegenden Datentypen und zugehörigen Operationen sowie Regeln zur Definition und Verwendung von komplexeren Datentypen. Die Befolgung der Regeln des CTS ist eine der Voraussetzung dafür, dass die in einer Sprache formulierten Programme durch das VES ausgeführt werden können. Im Hinblick auf die Sprachinteroperabilität ist das CTS ein bedeutender Schritt, es definiert ein gemeinsames Typsystem für eine Menge von Sprachen. Fehlerträchtige Umwandlungen entfallen, ein `Integer` in Visual Basic .NET (VB.NET) [Mic06d] entspricht beispielsweise exakt einem `int` in C#. Ein gemeinsames Typsystem ist für den angestrebten Grad an Interoperabilität jedoch noch nicht ausreichend.

Die Common Language Specification definiert eine Teilmenge des Typsystems CTS und eine zusätzliche Menge von zu befolgenden Regeln. Programme in Sprachen, die CLS-kompatibel sind, können über die Sprachgrenze hinweg miteinander interagieren. Hierbei werden zwei Stufen der Interoperabilität unterschieden. Zum einen gibt es so genannte **consumer.** Dies sind Sprachen[52], mit denen es nur möglich ist, CLS-kompatible Implementationen in anderen Sprachen zu nutzen. Hierzu gehört insbesondere der Aufruf von Methoden oder das Erzeugen von Instanzen. Zum anderen gibt es **extender**; diese Sprachen ermöglichen sowohl die Nutzung CLS-kompatibler Implementationen als auch das Bereitstellen solcher für andere consumer oder extender. Es ist hierbei wichtig zu bemerken, dass eine Sprache sowohl CLS-kompatible als auch CLS-inkompatible Teile enthalten kann. Möchte

[51] Nach der ECMA hat auch die International Organization for Standardization (ISO) die CLI standardisiert.

[52] In [ECM05b] heißt es genauer „Sprachen oder Werkzeuge“, letztere sind hier aber nicht von Interesse.

man nun beispielsweise in einer solchen Sprache eine Komponente für die Nutzung durch andere Sprachen zur Verfügung stellen, so müssen die Schnittstellen dieser Komponente und die von diesen eventuell erwarteten Parameter oder gelieferten Rückgabewerte CLS-kompatibel sein. Diese Anforderungen gelten nicht für die Teile der Implementation, welche nicht zur Schnittstelle gehören.

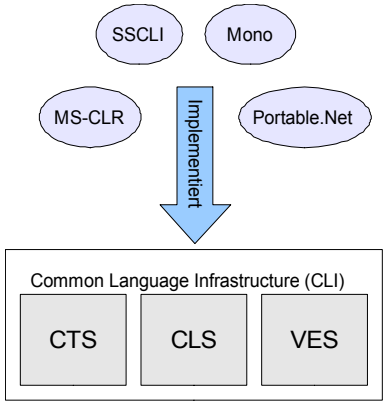


Abb. 5.1: Der Zusammenhang zwischen Implementationen und Standard

Unter der Bezeichnung Rotor bzw. Shared Source CLI Implementation (SSCLI) wird neben der CLR des .NET-Frameworks eine weitere Implementation der CLI durch Microsoft zur Verfügung gestellt, von dieser ist jedoch im Gegensatz zur kommerziellen Variante auch der Quellcode verfügbar. Detaillierte Informationen zu dieser Implementation finden sich in [SNS03]. Neben der SSCLI existieren mit dem Mono-Projekt [Mon06] und dem DotGNU-Projekt [Dot06] zwei weitere quelloffene Realisierungen des ECMA Standards. Diese drei Implementationen sind auch außerhalb der Microsoft Windows Plattform verfügbar, so dass mit CLI-kompatiblen Programmiersprachen betriebssystemübergreifende Softwareentwicklung prinzipiell[53] möglich ist. In Abb. 5.1 ist der Zusammenhang zwischen diesen Implementationen und dem Standard im Überblick dargestellt.

Die Standardisierung durch die ECMA und die Verfügbarkeit freier Implementationen sorgen dafür, dass es sich bei der CLR des .NET-Frameworks nicht um eine proprietäre Technologie handelt, die von einem einzelnen Hersteller kontrolliert wird.

Im Rahmen der CLI wird auch die so genannte **Common Intermediate Language** (CIL) definiert. Das Ziel, eine gemeinsame Laufzeitumgebung für eine Vielzahl von Sprachen zur Verfügung stellen zu können, wird durch die Definition einer gemeinsamen Zwischensprache, der CIL, erreicht. Ein Compiler erzeugt aus der Datei in der jeweiligen Sprache keinen maschinenspezifischen ausführbaren Code, sondern eine Repräsentation in der Zwischensprache. Für diese wird durch die Laufzeitumgebung ein **Just In Time-Compiler** (JIT) bereitgestellt, welcher die Zwischensprache in ausführbaren Maschinencode übersetzen und ausführen kann[54].

Folgendes Beispiel verdeutlicht die Möglichkeiten der Interoperabilität zwischen den Sprachen innerhalb des .NET-Frameworks. Verwendet werden in diesem Beispiel neben der Programmiersprache C# die beiden Sprachen Visual Basic .NET und J# [Mic06e]. In der Sprache VB.NET wird

[53] Hierbei müssen die Unterschiede der Zielbetriebssysteme, wie beispielsweise unterschiedliche Konventionen der Dateisysteme, berücksichtigt werden.

[54] Im .NET-Framework existiert die Möglichkeit, für Anwendungsszenarien, in welchen die Verwendung des JIT-Compilers nicht performant genug wäre, eine maschinenspezifische, ausführbare Datei zu erzeugen. Diese kann ohne einen erneuten Einsatz des JIT-Compilers ausgeführt werden. Eine detailierte Behandlung dieses Themas und auch der möglichen Nachteile findet sich in [Ric06].

eine Klasse `Person` mit einigen Attributen implementiert. Eine weitere Klasse wird in der Programmiersprache J# implementiert, diese stellt eine statische Methode `MakeSentence()` bereit. Diese Methode erwartet ein Objekt vom Typ `Person` – obgleich diese Klasse in VB.NET realisiert ist – und liefert einen beschreibenden Satz über die Person aus den Attributen des übergebenen Person-Objekts als `String` zurück. In C# wird eine Klasse `Mitarbeiter` als Spezialisierung der Klasse `Person` deklariert – die Sprachinteroperabilität innerhalb des .NET-Frameworks erlaubt Vererbungsbeziehungen über Sprachgrenzen hinweg. Das Hauptprogramm wird ebenfalls in C# entwickelt und verwendet die Methode `MakeSentence()` aus der in J# realisierten Klasse, um den Satz zu einem Personen- und zu einem Mitarbeiterobjekt zu bestimmen und gibt diesen auf der Konsole aus. Dies alles ist ohne Typumwandlungen oder eine spezielle, aufwendige Aufrufsyntax möglich. Die Nutzung der Methoden und Klassen in den unterschiedlichen Sprachen gestaltet sich aus Sicht des Nutzers vollkommen transparent. Das Diagramm Abb. 5.2 verdeutlicht diese Zusammenhänge, gestrichelte Linien zeigen eine Abhängigkeit an.

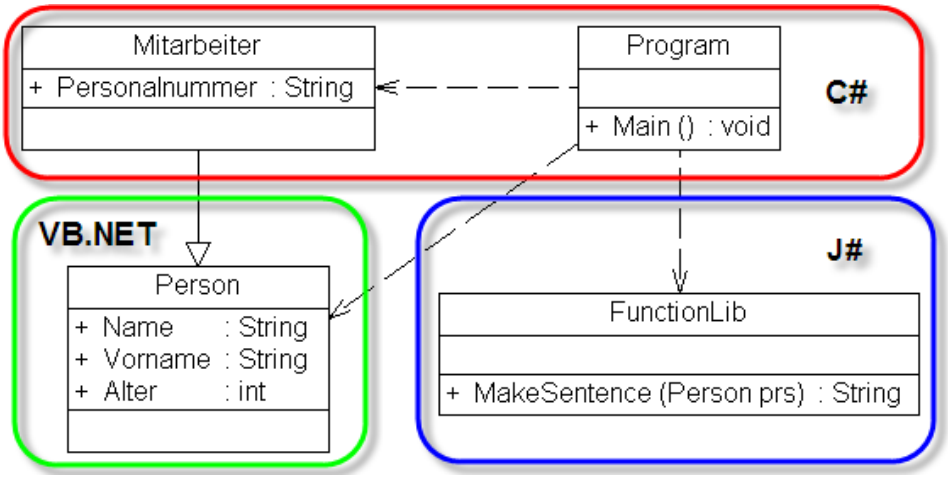


Abb. 5.2: Sprachinteroperabilität im .NET-Framework

Über die angesprochenen Festlegungen hinaus bestimmt die CLI auch eine grundlegende Klassenbibliothek und definiert die binären Formate der Ausgabedateien eines CLI-kompatiblen Compilers.

Die von Anfang an für den Entwurf des .NET-Frameworks bzw. des zugehörigen Standards vorhandene Zielsetzung der sprachübergreifenden Interoperabilität ist im Hinblick auf den Einsatz als Komponentenframework – wie in Kapitel 3 angesprochen – eine wünschenswerte Eigenschaft. Durch die Verfügbarkeit von Implementationen der Common Language Infrastructure auf verschiedenen Plattformen ist die Situation sogar noch bedeutend besser. Da wie bereits erwähnt sowohl die Common Intermediate Language als auch das Format der Dateien festgelegt ist, ist es in der Tat möglich, eine mit dem C#-Compiler des Mono-Projektes unter Linux übersetzte Bibliothek in der von Microsoft implementierten CLR unter Windows zu nutzen. Einschränkungen ergeben sich hier nur durch die verfügbaren Bibliotheken. Diese Problematik beschränkt sich jedoch auf nicht durch den CLI Standard definierte Bibliotheken.

Da Komponenten im Sinne der Definition aus Kapitel 3 als Strukturierungsmittel, Baustein und Verteilungseinheit fungieren sollen, werden nun die dafür notwendigen Konzepte des .NET-Frameworks dargestellt.

5.3 Assemblys

Um die Konzeption des Assemblys zu motivieren und die damit einhergehenden Vorteile erläutern zu können, ist es sinnvoll, zuerst einen Blick auf die vorhergehende Komponententechnologie auf der Windows-Plattform zu werfen. Die Ausführungen im Folgenden sollen keine vollständige Darstellung dieser Ansätze sein, sondern beschränken sich auf eine Verdeutlichung der Defizite des Component Object Model (COM), welche die Entwicklung des Assemblykonzeptes beeinflusst haben. Für eine weitergehende Betrachtung des Component Object Model sei auf [GT00] und [WK94] verwiesen.

COM definiert einen binären Standard, der die Interaktion von Komponenten ermöglicht. Dies ist auch für Komponenten möglich, die in unterschiedlichen Programmiersprachen entwickelt wurden. Es existiert aber anders als im CLI-Standard kein gemeinsames Typmodell. In der Folge kann ein Aufruf von Funktionalität über Sprachgrenzen hinweg entsprechende Konvertierungen von Parametern beziehungsweise Rückgabewerten erfordern.

Als Verteilungseinheit dienen im Falle von COM ausführbare Dateien. Hauptsächlich sind dies so genannte Dynamic Link Libraries (DLLs). Falls mehrere solcher Dateien im Rahmen einer einzelnen Applikation verwendet werden, so stellen diese zwar eine logische Einheit dar, während der Installation und im späteren Verlauf der Verwendung müssen sie jedoch einzeln behandelt werden. Aus diesem Umstand resultiert eine Reihe von Defiziten (vgl. [Loe03] [Mic06f]). So müssen beispielsweise alle Dateien einzeln auf dem Zielsystem registriert werden und bei einer späteren Deinstallation der Anwendung entfernt werden. Wenn – wie vielfach geschehen – die Realisierung der Installationsprogramme nachlässig erfolgt, bleiben so genannte verwaiste Dateien nach der Deinstallation zurück. Auch das Konfigurieren von Sicherheitseinstellungen und die Versionsverwaltung sind an einzelnen DLLs und nicht an der logischen Einheit der gemeinsam verteilten DLLs orientiert.

Bei der Versionsverwaltung im Component Object Model kommt hinzu, dass es nicht möglich ist, unterschiedliche Versionen einer DLL nebeneinander zu verwenden. Wenn nun eine Anwendung A die Version 1.0 einer DLL verwendet und die später installierte Anwendung B diese mit einer anderen Version überschreibt, so besteht das Risiko von Inkompatibilitäten durch ein geändertes Verhalten der Bibliothek. Eine Änderung der einmal publizierten Schnittstellen einer DLL mit einem Versionswechsel ist in COM nicht möglich, jede neue Version einer Bibliothek muss zu den vorherigen abwärtskompatibel sein.

Ein potenzielles Sicherheitsrisiko entsteht aus der alleinigen Verwendung von GUIDs[55] zur Identifikation von Komponenten im Component Object Model. Da diese bekannt sein müssen, um eine Komponente zu verwenden, ist es möglich, eine veränderte Variante der Komponente mit derselben GUID zu erstellen und auf diese Weise Code einzuschleusen. Die GUID garantiert zwar Eindeutigkeit als Bezeichner einer Komponente, ist aber kein ausreichender Mechanismus um Authentizität und Integrität der Komponente sicherzustellen.

Diese Unzulänglichkeiten waren der Ausgangspunkt für die Entwicklung der Assemblys. Ein Assembly kann aus einer einzelnen Datei bestehen oder es kann mehrere Dateien zu einer logischen Einheit zusammenfassen (engl. to assemble – zusammenfassen). Die Dateien eines Assemblys werden als Module bezeichnet. Dabei werden die Attribute, die zur Gesamtheit der Dateien gehören, von den einzelnen Dateien getrennt und nur einmal innerhalb des Assemblys im so genannten Manifest gespeichert. Jedes Assembly besitzt genau ein Manifest. Dieses legt den Namen des Assemblys fest und gibt dessen Version an. Darüber hinaus enthält das Manifest Lokalisierungs- und Sicherheits-

[55] Global Unique Identifier, dabei handelt es sich um eine hexadezimal geschriebene 128-Bit Ganzzahl. Weitere Informationen finden sich in [LMS05].

informationen. Da ein Assembly aus mehreren physischen Dateien bestehen kann, werden im Manifest gegebenenfalls auch Verweise auf alle zugehörigen Dateien festgehalten. Diese Verweise sind jedoch nur Einträge im Assemblymanifest und werden nicht durch Verweise im Dateisystem realisiert.

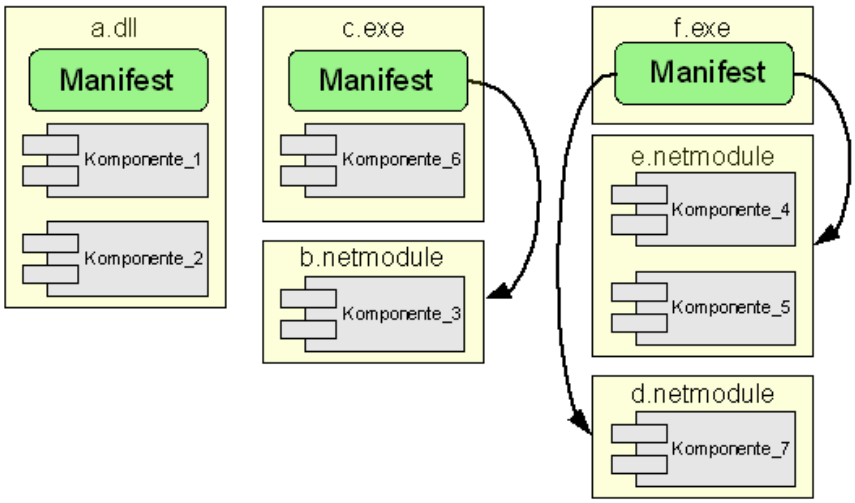


Abb. 5.3: Zusammenhang zwischen Assemblys und physischen Dateien

Wie die Grafik in Abb. 5.3 zeigt, gibt es diverse Möglichkeiten ein Assembly auf mehrere physische Dateien aufzuteilen. Eine Säule in der Grafik ist jeweils ein eigenständiges und vollständiges Assembly. Das Zusammenfassen von Dateien zu einem Assembly ist unabhängig von der Referenzierung anderer Assemblys im Sinne von genutzten Bibliotheken. Diese Referenzierung wird zwar auch im Manifest eines Assemblys eingetragen, führt aber nicht dazu, dass es eine gemeinsame Menge von Attributen für diese Assemblys gibt.

Die Abbildung zeigt mehrere Komponenten in einem Assembly. Dies entspricht der in der Literatur zum .NET-Framework verbreiteten Sichtweise, in der eine Klasse als Komponente gesehen wird (siehe z. B. [Loe03]). Die Zusammenfassung mehrerer Komponenten in einem Assembly führt – unabhängig davon, ob es sich bei einer Komponente um einzelne Klassen oder eine Menge interagierender Klassen handelt – dazu, dass diese Komponenten keine eigenständigen Verteilungseinheiten mehr darstellen. Folglich ist eine solche Zusammenstellung nur in den Fällen sinnvoll, in denen die Komponenten als unteilbare Einheit betrachtet werden. In diesem Fall sollte zumindest konzeptuell die Zusammenfassung der Einzelkomponenten als Komponente betrachtet werden.

In Abschnitt 3.2.2 wurde die Eigenschaft einer Komponente, sich hinsichtlich der Verwendung wie eine Black-Box einsetzen zu lassen, angesprochen. Auch Assemblys unterstützen dieses Konzept. Neben den Metadaten im Manifest, welche das gesamte Assembly beschreiben, existieren Metadaten, welche die enthaltenen Typen beschreiben. Diese sind jeweils in dem Modul enthalten, welches auch den zugehörigen CIL-Code enthält. Diese Metadaten lassen sich auch durch Programme über einen als Reflexion bezeichneten Mechanismus auslesen. Das Format der Metadaten ist ebenfalls durch den CLI-Standard vorgegeben. Durch den Einsatz der Reflexionsmechanismen ist es möglich, die von einem Assembly bereitgestellten Typen zu verwenden, ohne dass diese vor der Laufzeit einer Anwendung bekannt sein müssen. Auf dieser Basis lassen sich Plug-In-Mechanismen realisieren, um Applikationen anpassbar und erweiterbar zu machen. Im Rahmen der Entwicklung des im Verlauf

dieser Arbeit erstellten Prototyps wurde ausgiebig von diesen Möglichkeiten Gebrauch gemacht. Näheres hierzu findet sich in Kapitel 6, Aspekte der Implementation in C#.

Neben CIL-Code und Metadaten können die Module eines Assemblys auch so genannte Ressourcen enthalten. Dabei handelt es sich um Daten im weitesten Sinne; Icons, Grafiken und Stringressourcen sind Beispiele hierfür.

Die Möglichkeit, mehrere Dateien in einem Assembly zusammenzufassen, adressiert das eingangs angesprochene Problem, einer logischen Verteilungseinheit gemeinsame Attribute zuweisen zu können. Um die Versionskonflikte zu beheben, die entstehen, wenn von einer gemeinsam genutzten Bibliothek mehrere Versionen existieren, sind zusätzliche Mechanismen nötig. Zu diesem Zweck wird eine Differenzierung der Assemblys bezüglich ihrer Verwendung vorgenommen.

Assemblys werden in private und in gemeinsam genutzte unterteilt, je nachdem, ob sie nur zur Verwendung im Rahmen einer einzelnen Applikation gedacht sind oder als gemeinsame Ressource für alle Anwendungen der CLR zur Verfügung stehen sollen. Private Assemblys werden im Verzeichnis der Anwendung gespeichert und es ist keine Registrierung wie beispielsweise bei COM-Komponenten notwendig. Gemeinsam genutzte Assemblys (shared Assemblys) werden in einem zentralen Verzeichnis, dem so genannten **Global Assembly Cache** (GAC) abgelegt. Dieser erlaubt es, mehrere Versionen eines gemeinsam genutzten Assemblys nebeneinander abzulegen und zu verwenden. Diese Eigenschaft wird als **side-by-side execution** bezeichnet. Damit sind die bei DLLs üblichen Versionskonflikte nicht mehr möglich. Wenn eine Anwendung eine neuere Version eines Assemblys in den GAC einbringt, so bleiben alle Anwendungen, welche eine frühere Version verwenden, hiervon unberührt.

Damit ein Assembly im GAC abgelegt werden kann, benötigt es einen so genannten **starken Namen** (strong name). Der weiter oben angesprochene Name eines Assemblys wird in der angelsächsischen Literatur auch als **friendly name** bezeichnet. Hierbei handelt es sich um einen normalen Bezeichner (z. B. *MeinAssembly*). Der starke Name eines Assemblys wird hingegen auf Basis eines Public Key-Verschlüsselungsverfahrens erzeugt. Dadurch wird sichergestellt, dass ein Assembly – zusammen mit der Versionsnummer – eindeutig indentifiziert werden kann. Da zusätzlich eine kryptographische Signatur des Manifestes erzeugt wird, kann auch die Integrität des Assemblys sichergestellt werden. Im Falle von Assemblys mit mehreren Modulen enthält das Manifest auch Hashwerte für diese Module, diese werden also ebenfalls beim Erzeugen der Signatur des Manifestes berücksichtigt.

An dieser Stelle muss darauf hingewiesen werden, dass in den Client-Assemblys[56] einer Anwendung zwar die Versionen der referenzierten privaten Assemblys festgehalten werden, die CLR unterschiedliche Versionsnummern jedoch im Falle von Assemblys ohne starken Namen ignoriert.

Zum Abschluss der Betrachtungen sei noch auf einen interessanten Punkt im Zusammenhang mit gemeinsam genutzten Assemblys hingewiesen. Da beim Referenzieren eines Assemblys auch die Version angegeben wird, ist eine Änderung an den Schnittstellen der durch das Assembly bereitgestellten Komponenten prinzipiell möglich. Eine Anwendung, die eine ältere Version der Schnittstellen benötigt, referenziert auch die entsprechende Version des Assemblys im GAC. So können keine Konflikte durch die neue Schnittstelle auftreten. Dies ist eine Abkehr von dem im Component Object Model notwendigen Prinzip, einmal veröffentlichte Schnittstellen nicht mehr zu verändern.

[56] Als Client-Assembly wird hier ein Assembly bezeichnet, welches durch ein weiteres, so genanntes Server-Assembly, bereitgestellte Funktionalität nutzt.

5.4 Klassen, Assemblys und Komponenten

Assemblys als Grundlage der Realisierung von Komponenten auf Basis des .NET-Frameworks wurden im vorangegangenen Abschnitt erläutert. Auch die zentralen Eigenschaften des Frameworks, die in diesem Zusammenhang eine Bedeutung haben, wurden vorgestellt. Offen geblieben ist der Punkt, wie eine Komponente realisiert wird, auf welche Art und Weise ihre Schnittstellen beschrieben werden und wie sie geladen und instanziert werden kann. Diese Fragen sind jedoch nur abhängig von der Auswahl einer konkreten Programmiersprache zu beantworten.

Die Sprachen des .NET-Frameworks besitzen objektorientierte (zum Beispiel C#, VB.NET) oder auch funktionale (im Fall von OCaml) Sprachkonstrukte und neben den offiziell von Microsoft unterstützten Sprachen existieren eine Vielzahl weiterer. Die folgenden Ausführungen beziehen sich auf C#, da in dieser Sprache die prototypische Umsetzung der erarbeiteten Ansätze vorgenommen wurde[57].

Da es sich bei C# um eine objektorientierte Sprache handelt, ist es möglich, Klassen und Interfaces zu definieren. Es existiert jedoch kein Sprachkonstrukt, welches explizit zur Deklaration einer Komponente bestimmt ist. Das Zusammenfassen von Klassen in einem Assembly, der erste Schritt zur Komponente, wird nicht durch Mittel der Sprache sondern durch den Übersetzungsvorgang gesteuert. Unabhängig davon, ob die Klassen während der Entwicklung in einer oder in mehreren Quelltext-dateien verteilt entwickelt werden und unabhängig davon, ob sie im selben Namensraum[58] liegen oder nicht, können Klassen durch entsprechende Einstellungen des Compilers beim Übersetzen in ein gemeinsames Assembly zusammengefasst werden.

Ein Assembly wiederum kann nicht als Ganzes – im Sinne einer Komponente, wie sie in Kapitel 3 definiert wurde – instanziert werden. Dies ist nur für die in ihm definierten Typen möglich. Das Instanzieren der Typen in einem Assembly kann einzeln und zu unterschiedlichen Zeitpunkten geschehen, es besteht keine Notwendigkeit alle gemeinsam zu instanzieren. Das Assembly bildet die Verteilungseinheit, ein wichtiges Charakteristikum einer Komponente.

Die Schnittstellen der Komponenten müssen durch Schnittstellen im Sinne der objektorientierten Programmierung realisiert werden. Dies bedeutet in der Folge auch, dass nicht die Komponente bzw. das Assembly die Schnittstelle implementiert sondern eine oder mehrere Klassen innerhalb des Assemblys. Ein Assembly kann zwar Interfaces enthalten und die in einem Assembly enthaltenen Klassen können Interfaces implementieren; das Assembly selbst ist hierzu jedoch nicht in der Lage.

Um eine Komponente zu entwickeln, die ein bestimmtes Interface realisiert, muss in C# also eine das Interface realisierende Klasse in einem Assembly gekapselt werden. Im Sinne der in dieser Arbeit verwendeten Komponentendefinition erlaubt dies jedoch nur die Realisierung von Export-schnittstellen. Ergänzend ist hier anzumerken, dass in C# spezielle Zugriffsrechte (markiert mit dem Schlüsselwort `internal`) existieren, um den Zugriff auf Klasseneigenschaften auf Klassen im selben Assembly einzugrenzen. Diese Möglichkeit ist wichtig zur Realisierung des Geheimnisprinzips auf

[57] Für die von Microsoft unterstützten Sprachen (C#, VB.NET, C++.NET) lassen sich diese Aussagen analog übertragen, dies gilt vermutlich auch für eine Vielzahl der von anderen Anbietern bereitgestellten Sprachen. Allerdings listet [Rit06] zum Zeitpunkt dieser Ausarbeitung 49 Sprachen bzw. Sprachgruppen für die .NET-Plattform auf, für einige davon existieren mehrere Realisierungen. Aufgrund dieser enormen Bandbreite kann hier keine Aussage getroffen werden, ob die für C# geltenden Aussagen allgemeingültig für alle Sprachen der .NET-Plattform sind.

[58] Ein Namensraum (Schlüsselwort `namespace`) ist ein C# Sprachfeature zur logischen Gruppierung von Klassen und Interfaces. Namensräume können hierarchisch geschachtelt (z. B. `System.Reflection`) werden, Bezeichner müssen innerhalb eines Namensraumes eindeutig sein.

Komponentenebene, da es hiermit möglich ist, bei Zugriffen von außerhalb des Assemblys eine Beschränkung auf entsprechende Schnittstellen zu erzwingen.

Es sei an dieser Stelle nochmals darauf hingewiesen, dass die Eigenschaft Verteilungseinheit zu sein ein zentraler Aspekt einer Komponente im Sinne der Definition aus Kapitel 3 ist. Die hiervon abweichende Betrachtungsweise, die in Abb. 5.3 dargestellt ist, ist durch den Umstand motiviert, dass Klassen die Einheit der Instanzierung und Assemblys die Einheit für die Verteilung sind. Eine Komponente im Sinne der konzeptuellen Definition dieser Arbeit kann zwar realisiert werden, dies erfordert jedoch an einigen Stellen entsprechende Entwurfsentscheidungen und wird durch das Framework nicht erzwungen.

Der einzige in der Sprache C# bzw. dem .NET-Framework vorhandene Ansatz, der auf die Realisierung von Importschnittstellen bzw. die explizite Kenntlichmachung aller externen Abhängigkeiten eines Assemblys abzielt, ist im Assemblymanifest enthalten. Ein Assembly führt in seinem Manifest alle von ihm referenzierten Assemblys auf. Dadurch sind Abhängigkeiten, die zur Übersetzungszeit durch den Compiler erfasst werden können, benannt. Darüber hinausgehende Abhängigkeiten, die erst zur Laufzeit existieren, sind nicht im Assemblymanifest verzeichnet und müssen durch einen anderen Mechanismus beschrieben werden. Ein Beispiel für solche Abhängigkeiten wäre das Laden von Initialisierungsinformationen aus einer Datenbank. Dabei handelt es sich um eine Kontextabhängigkeit, welche im Assemblymanifest nicht erkennbar wäre. An dieser Stelle werden die Anforderungen der Definition durch die vom .NET-Framework bereitgestellte Funktionalität nicht unterstützt.

In den Szenarien, in denen eine vollständige Beschreibung der Kontextabhängigkeiten einer Komponente existieren muss – dies ist beispielsweise bei ihrer Weitergabe an einen zukünftigen Nutzer der Fall – müssen entsprechende Möglichkeiten selbst realisiert werden. Dies kann im einfachsten Fall in Form einer textuellen Dokumentation geschehen, wenn die Komponente innerhalb eines Softwareprojektes Verwendung finden soll. Je nach Einsatzszenario kann auch eine Beschreibung vonnöten sein, die sich automatisch verarbeiten lässt. Für die Eignung des .NET-Frameworks zur Realisierung der prototypischen Implementation im Rahmen dieser Arbeit ist dieser Punkt jedoch nur von nachrangiger Bedeutung.

5.5 Zusammenfassung

Das .NET-Framework und die Sprache C# waren zum Zeitpunkt der anfänglichen Untersuchungen zu dieser Arbeit sehr neu und in vieler Hinsicht viel versprechend. Inzwischen ist die zweite Version der Sprache C# standardisiert, eine dritte befindet sich im Entwicklungsprozess [Mic06g].

Die durch die CLI beschriebene Sprachinfrastruktur erlaubt mit einigen Abstrichen, die sich aufgrund des Fehlens rein komponentenorientierter Spracheigenschaften ergeben, die Umsetzung des Komponentenkonzeptes, wie es in Kapitel 3 dieser Arbeit definiert wird. Dabei müssen solche Eigenschaften der Definition, welche Entwurfsziele oder Entwurfskriterien beinhalten, bei der Realisierung der Komponenten durch den Entwickler beachtet werden. An dieser Stelle ist nur eine Unterstützung durch die zugrunde liegende Technologie möglich.

Assemblys stellen eine in jeder CLI-Implementierung ausführbare Verteilungseinheit dar, eine wichtige Eigenschaft für die Portabilität von Anwendungen. Auch aus mehreren Modulen bestehende Assemblys sind eine unteilbare Einheit hinsichtlich Versionierung und Sicherheit. Durch die in den Modulen enthaltenen Metadaten zur Beschreibung der im jeweiligen Modul enthaltenen Typen und die

Metadaten über das gesamte Assembly im Manifest sind Assemblys weitgehend selbstbeschreibend. Dies erlaubt eine Black-Box-Nutzung.

Die Sprache C# bietet – wie andere .NET-Sprachen auch – Sprachkonstrukte, die den Zugriff auf Assemblys zur Laufzeit ermöglichen, näheres hierzu findet sich in Abschnitt 6.2, Dynamisches Laden von Typen in C#.

6 Aspekte der Implementation in C#

„Als es noch keine Computer gab, war das Programmieren noch relativ einfach. Als es dann ein paar leistungsschwache Computer gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Computer haben, ist auch das Programmieren zu einem gigantischen Problem geworden.“
Edsger Wybe Dijkstra, niederländischer Informatiker, 1930-2002

In diesem Kapitel werden einige ausgewählte Aspekte der Umsetzung des Entwurfs für das IDE-Framework in der gewählten Implementationssprache C# vorgestellt. Dabei werden lediglich solche Teilbereiche der Implementation angesprochen, die besonders zentral, interessant oder problematisch sind, da eine vollständige Abhandlung der vorgenommenen Implementation über den Rahmen dieser Arbeit hinausgehen würde.

Vorgestellt werden zunächst einige Implementationstechniken, die eine Umsetzung komponentenorientierter Ansätze unter Verwendung der Programmiersprache C# erlauben. Im weiteren Verlauf dieses Kapitels wird dann ein einfaches Modell vorgestellt, welches mit dem Ziel des Nachweises der grundsätzlichen Umsetzbarkeit der Überlegungen aus Kapitel 4 implementiert wurde. Neben diesem Modell werden ausgewählte Aspekte der Umsetzung der Dienste im Cobamos-Framework dargestellt. Den Abschluss des Kapitels bildet der Entwurf einer Kommandokomponente zur Realisierung der zur Speicherung von Modellzuständen notwendigen Funktionalität, dieser Entwurf verdeutlicht die Möglichkeiten, die sich aus einer Interaktion von Diensten und Kommandokomponenten ergeben.

6.1 Einleitung und Motivation

Das in Kapitel 4 entworfene Framework gibt naturgemäß nur einen Rahmen für weitere Entwicklungen vor. Der vorgestellte Entwurf ist daher konfigurier- bzw. erweiterbar. Dies soll das Cobamos-Framework durch entsprechende Anpassungen in möglichst vielen Szenarien nutzbar machen. Um zukünftige Projekte möglichst weit zu entlasten, wurde der für eine Entwicklungsumgebung relevante Teil des Frameworks in Form einer – auch ohne Erweiterungen – lauffähigen Anwendung realisiert.

Das Vorhandensein einer lauffähigen Anwendung als Ausgangspunkt erleichtert ein iteratives und inkrementelles Vorgehen zur Realisierung von Erweiterungen und Anpassungen des Frameworks beziehungsweise einer darauf basierenden Entwicklungsumgebung. Idealerweise ist am Ende jedes Erweiterungsschrittes eine lauffähige Zwischenversion vorhanden, welche getestet werden kann. Ein solches Vorgehensmodell wurde auch während der Entwicklung der IDE, der bereitgestellten Dienste und des auf dieser Basis realisierten Programmiermodells (siehe Kapitel 7 und 8) verfolgt.

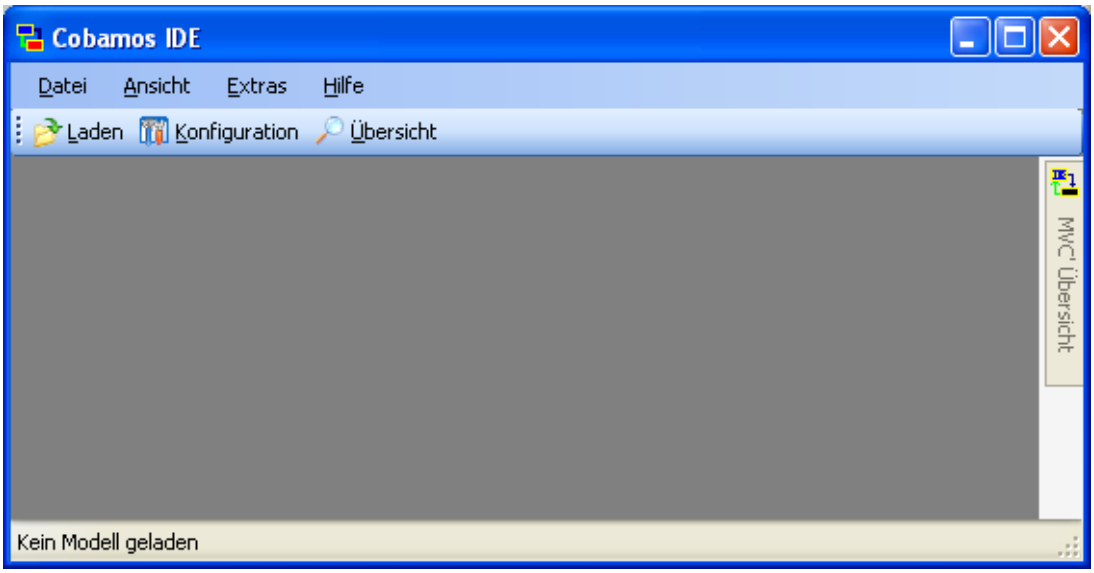


Abb. 6.1: Die Cobamos-IDE ohne zusätzliche Kommandokomponenten und ohne geladenes Modell

Die „leere“ Entwicklungsumgebung, die mit dem Framework bereitgestellt wird, ist in Abb. 6.1 dargestellt. Neben den verschiedenen Diensten, welche jedoch keine Entsprechung in der Nutzungsoberfläche haben, werden durch diese Anwendung die benötigte Funktionalität zur Integration von nach dem MVC'-Muster realisierten Programmiermodellen und von dynamischen Komandokomponenten sowie die Fensterverwaltung für eine MDI-Anwendung zur Verfügung gestellt. Bis auf die Fensterverwaltung verwenden alle diese Funktionen die in C# beziehungsweise dem .NET-Framework vorhandenen Möglichkeiten zum dynamischen Laden von Assemblys oder genauer der darin definierten Typen zur Laufzeit einer Anwendung. Darüber hinaus sind diese Möglichkeiten – wie in Kapitel 5 bereits angesprochen – von erheblicher Bedeutung für die Eignung der .NET-Technologie zur Realisierung komponentenbasierter Anwendungen. Bei den folgenden Ausführungen wird daher ein Schwerpunkt auf die Darstellung der Teile der vorgenommenen Implementation gelegt, in denen diese Konzepte Anwendung finden.

6.2 Dynamisches Laden von Typen in C#

In diesem Abschnitt werden die Möglichkeiten von C# beschrieben, welche im Rahmen der prototypischen Implementierung verwendet werden, um das in Abschnitt 4.3.2 angesprochene dynamische Laden von Komponenten zur Laufzeit einer Anwendung zu realisieren. Die in C# vorhandene Technologie bildet dabei nicht exakt die konzeptuelle Sicht ab, die in Kapitel 3 dargestellt wurde. Während konzeptuell eine Komponente als Ganzes geladen und instanziert wird, müssen in C# zwei Sprachkonstrukte in diesem Prozess unterschieden werden. Die bereits erwähnten Assemblys stellen das Kapselungskonstrukt hinsichtlich des Ladens dar, instanziert werden die enthaltenen Typen. Dementsprechend kann ein Assembly mehrere instanzierbare Typen enthalten.

Grundsätzlich lassen sich beim dynamischen Laden von Komponenten zwei Fälle unterscheiden, abhängig davon, ob die Schnittstelle zum Zugriff auf die Funktionalität zur Entwurfszeit bekannt ist oder nicht. In den Szenarien, die bei der Implementation des Cobamos-Frameworks eine Rolle spielen, ist zumindest eine allgemeine Schnittstelle bekannt, welche die zu ladenden Typen implementieren. Wäre dies nicht der Fall, müssten die entsprechenden Informationen zur Laufzeit ermittelt werden und die notwendigen Aufrufe dynamisch erstellt werden. Die Möglichkeiten hierfür sind in C# vorhanden,

da ein solches Vorgehen jedoch einen höheren Aufwand erfordert und sich negativ auf die Performanz der resultierenden Anwendung auswirkt, ist ein Einsatz nur dann sinnvoll, wenn dieses hohe Maß an Flexibilität wirklich benötigt wird.

Der Namensraum `System.Reflection` [Mic06h] der C#-Klassenbibliothek stellt unter anderem die für das Laden von Assemblys notwendigen Klassen zur Verfügung. Die Klasse `Assembly` besitzt eine Reihe statischer Methoden um Assemblys auf unterschiedliche Art und Weise, zum Beispiel über einen Dateinamen oder aus einem Datenstrom, zu laden. Diese Methoden liefern im Erfolgsfall eine Instanz der Klasse `Assembly` zurück. Neben Methoden, um die zu einem Assembly gehörigen Ressourcen oder Module laden zu können, besitzt die Klasse `Assembly` auch Methoden, die es erlauben, auf die in einem Assembly vorhandenen Typen zuzugreifen. Da sich diese Methoden auf ein konkretes Assembly beziehen, handelt es sich um Instanzmethoden.

```
(1)   public Type[] GetImplementorsFromAssembly(string assembly, string
                                                          interfaceToImplement)
(2)   {
(3)   [...]
(4)      Assembly _asm = Assembly.LoadFrom(assembly);
(5)   [...]
(6)
(7)      Type[] allTypes = _asm.GetTypes();
(8)      ArrayList returnTypes = new ArrayList();
(9)      foreach (Type t in allTypes)
(10)        if (t.GetInterface(interfaceToImplement)!=null && !t.IsAbstract)
(11)              returnTypes.Add(t);
(12)     return (Type[])returnTypes.ToArray(typeof(Type));
(13)  }
```

Codebeispiel 6.1: Das Laden von Typen aus einem Assembly in C#

Das Codebeispiel 6.1 zeigt die Methode `GetImplementorsFromAssembly()`, diese dient zum Ermitteln von Typinformationen aus einem Assembly. Der Quelltext der gezeigten Methode stammt aus dem für das Framework implementierten Dienst `AssemblyLoaderService`, welcher die gesamte Funktionalität zum dynamischen Laden von Assemblys kapselt und über die Servicearchitektur innerhalb des Cobamos-Frameworks zur Verfügung stellt. Die hier verwendete Vorgehensweise lässt sich jedoch für ähnliche Problemstellungen verallgemeinern.

Die Methode realisiert das Laden eines Assemblys und das anschließende Durchsuchen dieses Assemblys nach allen Typen, die ein Interface mit dem im Parameter `interfaceToImplement` (siehe Zeile 1) übergebenen Bezeichner realisieren. Der Anfang der Methode wurde sinngemäß vereinfacht. Hier findet sich im Original zum einen das Laden der Informationen über das physikalische Assembly in eine lokale Variable `_asm` vom Typ Assembly. Zum anderen ist eine Fehlerbehandlung für den Fall, dass das physikalische Assembly mit dem übergebenen Namen nicht gefunden werden kann, an dieser Stelle integriert. Diese basiert auf anderen, bisher noch nicht besprochenen, Diensten.

Zunächst werden (siehe Zeile 7) alle Typen aus dem geladenen Assembly ermittelt. Für diese Aufgabe steht die Methode `GetTypes()` der Klasse `Assembly` zur Verfügung, diese liefert ein Array mit Instanzen der Klasse `Type`. Diese Klasse aus dem Namensraum `System` enthält alle zur Laufzeit verfügbaren Informationen über einen Typ. Neben dem Bezeichner des Typs, seiner Sichtbarkeit und Ähnlichem, sind auch Informationen über die Methoden und Attribute des Typs vorhanden. Mit einem zu einem Typ gehörigen Type-Objekt ist es zudem möglich, eine Instanz dieses Typs zu erzeugen. Da in den Anwendungen innerhalb des Frameworks nur die Typen benötigt werden, die ein bestimmtes Interface

– beispielsweise `IService` im Falle eines Dienstes, `ICobamosCommand` im Falle eines nachladbaren Kommandos oder `IView`, falls es sich um eine Sicht zu einem Programmiermodell handelt – implementieren, wird ein zweites Array mit den entsprechenden Typen aufgebaut (siehe Zeilen 8-11). Die Methode `GetInterface()` der Klasse `Type` liefert ein `Type`-Objekt zu dem übergebenen Interface, wenn der Typ, zu dem sie aufgerufen wird, dieses Interface implementiert oder erbt. Da die ermittelten Typen später instanziert werden sollen, müssen auch die abstrakten Typen aussortiert werden. Dies lässt sich leicht durch eine Abfrage der Eigenschaft `IsAbstract` der den Typ beschreibenden `Type`-Instanz ermitteln.

Die Verwendung der Klasse `ArrayList` aus dem Namensraum `System.Collections` ist zweckmäßig, da im Vorhinein nicht klar ist, wie groß das Ergebnisarray sein wird. Diese Klasse stellt eine einem Array ähnliche Datenstruktur bereit, der Speicherplatz ist jedoch hinsichtlich der verwendeten Größe dynamisch. Da die Methode zum Umwandeln in ein Array als Rückgabetyp die abstrakte Klasse `Array` besitzt, ist eine zusätzliche Typumwandlung vor der Rückgabe notwendig.

Ein Problem tritt auf, wenn ein auf diese Art und Weise dynamisch geladenes Assembly zur Laufzeit der Anwendung durch eine andere Version ersetzt werden soll. Im obigen Codebeispiel 6.1 wird in Zeile 4 das Assembly geladen. Um nun eine neue Version des selben Assemblys laden zu können, müsste diese zuvor geladenen entfernt werden. Leider verfügt die Klasse `Assembly` nicht über eine entsprechende Methode. Um die benötigte Funktionalität dennoch realisieren zu können, ist der Umweg über die so genannte **Application Domain** (engl. Anwendungsdomäne) notwendig. Eine Anwendungsdomäne im .NET-Framework ist ein leichtgewichtiger Prozesstyp. Jede .NET-Anwendung läuft in einem eigenen Betriebssystemprozess; der Prozess ist für viele Aufgaben jedoch ein zu grobes Modularisierungskonzept und ist daher weiter in so genannte Anwendungsdomänen unterteilt. Ein Prozess besitzt dabei mindestens eine Anwendungsdomäne, kann aber beliebig viele enthalten. Diese können unabhängig voneinander erzeugt und beendet werden; wenn eine Anwendungsdomäne durch einen Fehler „abstürzt“ bleiben die übrigen Anwendungsdomänen des Prozesses hiervon unberührt. Hinsichtlich des Problems, ein einmal geladenes Assembly wieder entfernen zu können, ist die Tatsache wichtig, dass ein Assembly immer in einer bestimmten Anwendungsdomäne geladen wird. Für diese besteht die Möglichkeit, sie zur Laufzeit einer Anwendung vollständig zu entladen, dabei werden auch alle in diesem Kontext geladenen Assemblys entfernt. Dieser Umweg erlaubt es dann, ein Assembly durch eine neue Version zu ersetzen.

Das Erzeugen von Instanzen auf Basis der ermittelten Typinformationen ist der noch verbleibende Schritt zur Realisierung der benötigten Funktionalität. Dies wird im folgenden Abschnitt anhand des für das Nachladen der Kommandokomponenten zuständigen Teils der Anwendung erläutert.

6.3 Instanzierung am Beispiel des Nachladens von Kommandokomponenten

Der vorangegangene Abschnitt beschreibt anhand des im Cobamos-Framework realisierten Dienstes zum Laden von Typinformationen aus Assemblys die Möglichkeiten der Sprache C#. Auf Basis solcher Typinformationen ist es wie oben angesprochen möglich, Instanzen der entsprechenden Typen zu erzeugen. Dies geschieht im realisierten Framework unter anderem beim Laden der dynamischen Interaktionskomponenten (vergleiche Abschnitt 4.3.5). Wie bei der Schilderung des Entwurfs bereits erläutert, werden die Komponenten eines Verzeichnisses beim Start der Entwicklungsumgebung darauf geprüft, ob sie – oder genauer die in ihnen enthaltenen Klassen – das Interface `ICobamosCommand` implementieren. Die Klasse `CobamosCommandManager` ist die zentrale Klasse für diese Aufgaben im IDE-Framework. Sie ist als Singleton implementiert, da innerhalb des Frameworks zu einem

Zeitpunkt nur eine Instanz der Verwaltungsklasse für die dynamischen Kommandokomponenten existieren darf.

Das Ermitteln der in einem Verzeichnis vorhandenen Dateien ist eine Standardaufgabe und wird aus diesem Grund hier nicht näher behandelt. Für diese Dateien wird jeweils die in Codebeispiel 6.2 gezeigte Methode `LoadCommandsFromFile()` aufgerufen, in welcher die eigentliche Instanzierung durchgeführt wird.

```
(1)   public void LoadCommandsFromFile(string filename)
(2)   {
(3)      #if Debug
(4)         log.Log(TypeOfEntry.Debug, "Adding Commands from File: " + filename);
(5)      #endif
(6)
(7)      ArrayList commandsToLoad = new ArrayList();
(8)      AssemblyLoaderService asmLoader =
         (AssemblyLoaderService)ServiceManager.Services[typeof(AssemblyLoaderService)];
(9)      commandsToLoad.AddRange(asmLoader.GetImplementorsFromAssembly(filename,
                                                                commandInterface));
(10)     foreach (Type t in commandsToLoad)
(11)     {
(12)        ICobamosCommand command = (ICobamosCommand)Activator.CreateInstance(t);
(13)        this.AddComand(command);
(14)        #if Debug
(15)           log.Log(TypeOfEntry.Debug, "Added Command: " + command.ToString());
(16)        #endif
(17)     }
(18)  }
```

Codebeispiel 6.2: Das Instanzieren von CobamosCommands

Der Quellcode dieser Methode zeigt am Anfang und Ende des Quelltextes die Verwendung von Compilerdirektiven in C#, um während der Entwicklung für die Fehlersuche innerhalb der Anwendung notwendige Codeabschnitte aus der finalen Version entfernen zu können. Über die durch die Klasse `ServiceManager` realisierte Diensteverwaltung des Frameworks – diese ist wie die Klasse `CobamosCommandManager` als Singleton implementiert – wird eine Referenz auf eine Instanz des `AssemblyLoaderService` erfragt. Wie hier zu sehen ist, erfolgt die Identifikation der Dienste innerhalb der prototypischen Implementierung anhand des Typs des benötigten Dienstes, genauer durch ein den Typ beschreibendes Objekt der Klasse `Type`. Bei diesem Typ kann es sich auch um ein Interface oder eine abstrakte Klasse handeln. Der Operator `typeof` gehört zum Sprachumfang von C# und liefert den Typ der übergebenen Klasse. Der Servicemanager ist so realisiert, dass ein Zugriff auf die Dienste in der gezeigten Indexschreibweise möglich ist. Da der Servicemanager die Instanz des Dienstes nur mit dem Typ `IService` zurückliefert, ist vor der Verwendung eine Typumwandlung notwendig. Durch die Kapselung der zum Ermitteln der Typinformationen benötigten Funktionalität in einem Dienst ist an dieser Stelle kein Detailwissen über die hierfür verwendete Realisierung nötig. Die hierbei benutzte Methode `GetImplementorsFromAssembly()` ist die im vorherigen Abschnitt (siehe Codebeispiel 6.1) erläuterte.

Die ermittelten Typinformationen werden verwendet, um mithilfe der Klasse `Activator` des Namensraumes `System` Instanzen zu erzeugen. Ein Aufruf in der hier gezeigten Form setzt voraus, dass die Typen über einen parameterlosen Standardkonstruktor verfügen. Prinzipiell ist auch die

Verwendung von Konstruktoren mit Parametern möglich, diese sind aber hier nicht notwendig. Da die Signatur eines Konstruktors nicht durch ein Interface vorgegeben werden kann, wären für die Verwendung von parameterbehafteten Konstruktoren zudem entsprechende Konventionen über Art und Reihenfolge der Parameter nötig. Jede noch so kleine Abweichung würde zu einem Fehler im Aufrufcode zur Laufzeit der Anwendung führen. Angesichts der Absicht, die dynamisch ladbaren Teile der IDE durch verschiedene Entwickler realisieren zu lassen und auf dieser Basis Ergänzungen und Erweiterungen zu ermöglichen, sollte dieser Teil der Anwendung jedoch möglichst robust gestaltet werden. Benötigt ein Kommando oder ein Dienst bestimmte Daten oder Referenzen auf andere Objekte, so können diese nicht mit einem Konstruktor übergeben werden. Daher müssen entsprechende Methoden zum Setzen dieser Parameter im Interface deklariert werden. Dieses Vorgehen betont die Forderung nach expliziten Kontextabhängigkeiten, wie sie in Kapitel 3 für Komponenten aufgestellt wurde.

Die so erzeugte Instanz wird zu den bereits vorhandenen CobamosCommand-Instanzen hinzugefügt. Die Methode `AddCommand()` trägt die Instanz dabei nicht nur in eine interne Verwaltung ein, sondern erzeugt auch anhand der über das Interface `ICobamosCommand` ermittelten Informationen den Eintrag im Menü des Hauptfensters und gegebenenfalls auf der so genannten Werkzeugleiste. Der Quelltext dieser Methode ist in Codebeispiel 6.3 zu sehen.

```
(1)   public void AddComand(ICobamosCommand command)
(2)   {
(3)      this._commands.AddCommand(command);
(4)      if (command is IServiceConsumer)
(5)         (command as IServiceConsumer).SiteServiceManager = ServiceManager.Services;
(6)
(7)      if (this._toolbar != null && command.ShowOnToolbar)
(8)      {
(9)         ToolStripButton b = makeButton(command);
(10)        addToToolBar(b);
(11)     }
(12)
(13)     if (this._mainMenu != null)
(14)     {
(15)        ToolStripMenuItem m = makeMenuItem(command);
(16)        if (command.Position != null)
(17)           addToMenu(command.Position, m);
(18)        else
(19)           addToDefaultMenu(m);
(20)     }
(21)  }
```

Codebeispiel 6.3: Die Methode AddCommand der Klasse CobamosCommandManager

Interessant ist die Verwendung des Interfaces `IServiceConsumer`. Dieses dient zur Unterscheidung der Kommandokomponenten in zwei Gruppen. Ein Teil der realisierten Kommandos benötigt Zugriff auf das Dienstesystem des IDE-Frameworks. Es wäre prinzipiell möglich, dass die entsprechenden Realisierungen direkt auf die Singleton-Instanz der Klasse `ServiceManager` zugreifen, da diese frameworkweit verfügbar ist. Ein solches Vorgehen basiert zu einem gewissen Teil auf implizitem Entwurfswissen über die interne Realisierung der Diensteverwaltung; bei der Realisierung der Kommandos sollte jedoch eine möglichst lose Kopplung erreicht werden. Aus diesem Grund wurde das Interface `IServiceConsumer` als eine zusätzliche Rolle, die Realisierungen des Interfaces

`ICobamosCommand` einnehmen können, eingeführt. Das Interface deklariert eine Eigenschaft `SiteServiceManger` vom Typ der Schnittstelle `IServiceManager`. Dies ist das Interface, welches eine Klasse zur Diensteverwaltung im Cobamos-Framework implementieren muss.

In der in Codebeispiel 6.3 gezeigten Methode `AddCommand()` kann aufgrund dieser Realisierung recht einfach geprüft werden, ob die gerade bearbeitete Kommandokomponente Zugriff auf die Dienste benötigt. Dies reduziert sich auf die Frage, ob die Instanz vom Typ `IServiceConsumer` ist. Ist dies der Fall, so wird der Instanz eine Referenz auf die Diensteverwaltung übergeben.

Im verbleibenden Abschnitt der Methode werden – falls Referenzen auf die Werkzeugleiste bzw. das Hauptmenü vorhanden sind – die visuellen Repräsentationen der Kommandos in Form von Schaltflächen oder Menüeinträgen erzeugt und diese eingefügt. Die in Zeile 17 des Quelltextfragmentes aufgerufene Methode `addToMenu()` ist dafür zuständig, dass ein Menüeintrag an der richtigen Stelle eingetragen wird. Eine Positionsinformation einer Realisierung von `ICobamosCommand` – enthalten in der Eigenschaft `Position` – ist in der derzeitigen Realisierung ein String der Form „Menü1|Menü2|Menü3“, wobei die Tiefe durch die Implementation nicht beschränkt ist. Beim Eintragen wird nun zuerst der Menüpunkt „Menü1“ gesucht. Falls die Suche erfolglos endet, wird dieser Eintrag erzeugt und zum bestehenden Menü hinzugefügt. Dies wird für alle weiteren Stufen wiederholt, so dass es möglich wird, geschachtelte Menüstrukturen aufzubauen (siehe Abb. 6.2). Ein Problem stellen zunächst eventuell identische Positionsangaben unterschiedlicher Kommandos dar. Da die zur Realisierung der Menüstrukturen verwendete Bibliothek das Vorhandensein mehrerer Einträge mit gleichem Namen innerhalb einer Menüebene zulässt, wird hier jedoch die einfachste Strategie gewählt und ein weiterer Eintrag erzeugt. Die Methode `addToDefaultMenu()` fügt den übergebenen Eintrag in einem Standardmenü für dynamisch geladene Kommandos ein.

An dieser Stelle ist zu bemerken, dass die gewählte Form der Realisierung es erlaubt, mehrere Kommandos in einem gemeinsamen Assembly zusammenzufassen. Dies ist beispielsweise bei einer Menge von Standardkommandos für einen bestimmten Einsatzzweck des IDE-Frameworks sinnvoll, da es den Ladevorgang für die Kommandos und damit das Starten der Anwendung aufgrund der geringeren Anzahl notwendiger Dateioperationen beschleunigt. Ein solches Zusammenfassen führt – wie zuvor bereits angesprochen – jedoch dazu, dass die einzelnen Kommandos, oder genauer die sie implementierenden Typen, nicht mehr mit einer Komponente im Sinne der in Kapitel 3 dargelegten Definition gleichgesetzt werden können. Vielmehr handelt es sich bei dem Assembly um eine Komponente, welche mehrere Kommandos zur Verfügung stellt.

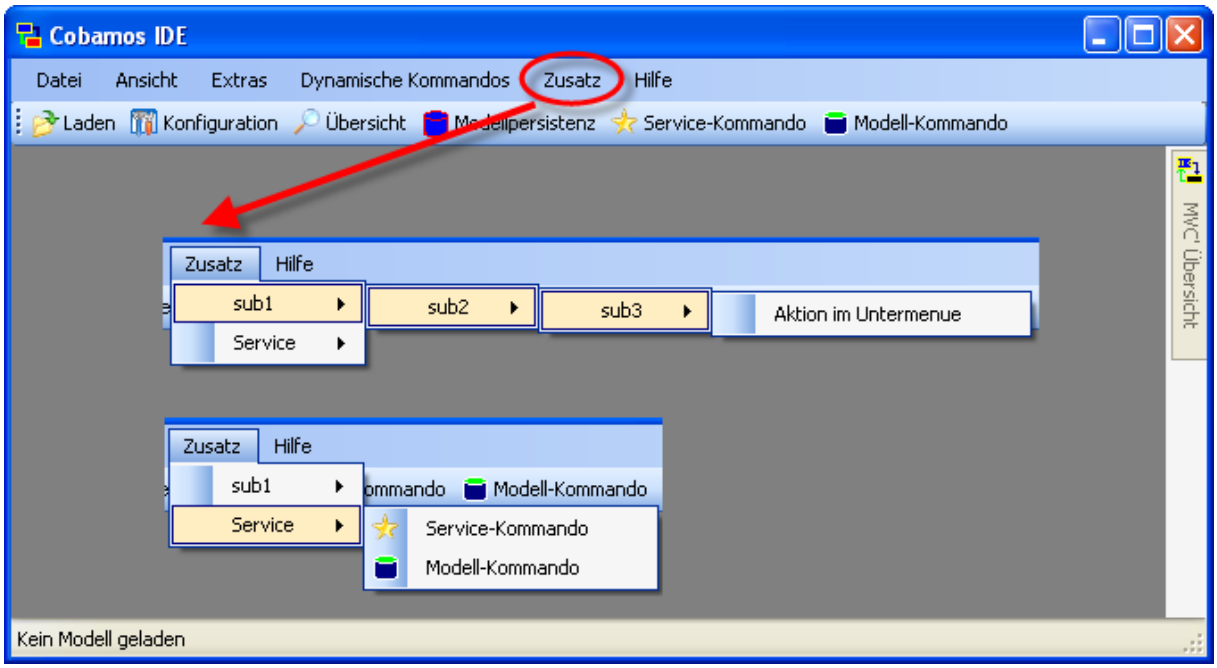


Abb. 6.2: Die Cobamos IDE mit geladenen Kommandos

In Abb. 6.2 ist die IDE mit geladenen Kommandos zu sehen. Der Menüpunkt „Dynamische Kommandos“ wird als Sammelmenü für alle Kommandos ohne Positionsangabe verwendet. Der Menüpunkt „Zusatz“ wurde neu erzeugt, da einige Kommandos Positionen in diesem Menü angegeben haben. Hier ist auch zu sehen, dass der Algorithmus in der Lage ist, geschachtelte Menüstrukturen zu erzeugen[59]. Einige der Kommandos werden auch auf der Werkzeugleiste des Hauptfensters angezeigt. Der angezeigte Name wird ebenfalls abgefragt, ebenso ein optionales Icon. Das gezeigte Service-Kommando ermittelt dieses über den RessourcenService, einen Dienst des Frameworks, das Modell-Kommando verwendet hierzu eine in das Assembly eingebettete Ressource.

Die hier beschriebene Funktionalität erlaubt eine Anpassung der IDE auf unterschiedliche Nutzergruppen oder Anwendungsfälle durch ein simples Austauschen oder Hinzufügen von Dateien. Das Hilfesystem ist aus diesem Grund ebenfalls durch eine Kommandokomponente realisiert. Damit es nicht zusammen mit anderen Kommandos versehentlich entfernt oder überschrieben wird, wird ein gesondertes Verzeichnis für diese Assemblys verwendet. Ein Erweitern der Hilfefunktionalität für Nutzer der Entwicklungsumgebung mit mehr Bedarf an Unterstützung ist somit leicht möglich.

6.4 Implementation eines Testmodells für den MVC'-Ansatz

Als weiterer Teilaspekt der Implementation der in Kapitel 4 entwickelten Konzepte soll in diesem Abschnitt die Realisierung des Ladens und die Verwaltung der Programmiermodelle und der zugehörigen Sichten besprochen werden.

Zum Entwickeln und Testen der Implementation und der Tragfähigkeit der Konzepte rund um das MVC'-Muster in einer realen Implementation ist ein geeignetes Testmodell mit entsprechenden Sichten nötig. Dabei müssen mehrere Sichten auf das Modell möglich sein und Änderungen der Modelldaten erfolgen, um die erfolgreiche Aktualisierung der Sichten testen zu können. Ein Modell, welches sich hierfür anbietet – auch wenn es nicht aus dem Bereich der Endnutzerprogrammierung stammt – ist eine Zeitanzeige. Das zugrunde liegende Modell ist wenig komplex und es ist in nahe

[59] Die Positionsangabe zum Kommando „Aktion im Untermenue“ lautete „Zusatz|sub1|sub2|sub3“.

liegender Weise möglich, unterschiedliche Sichten auf das Modell zu definieren. Damit ist es sehr gut zum Test des Frameworks und dessen Weiterentwicklung geeignet.

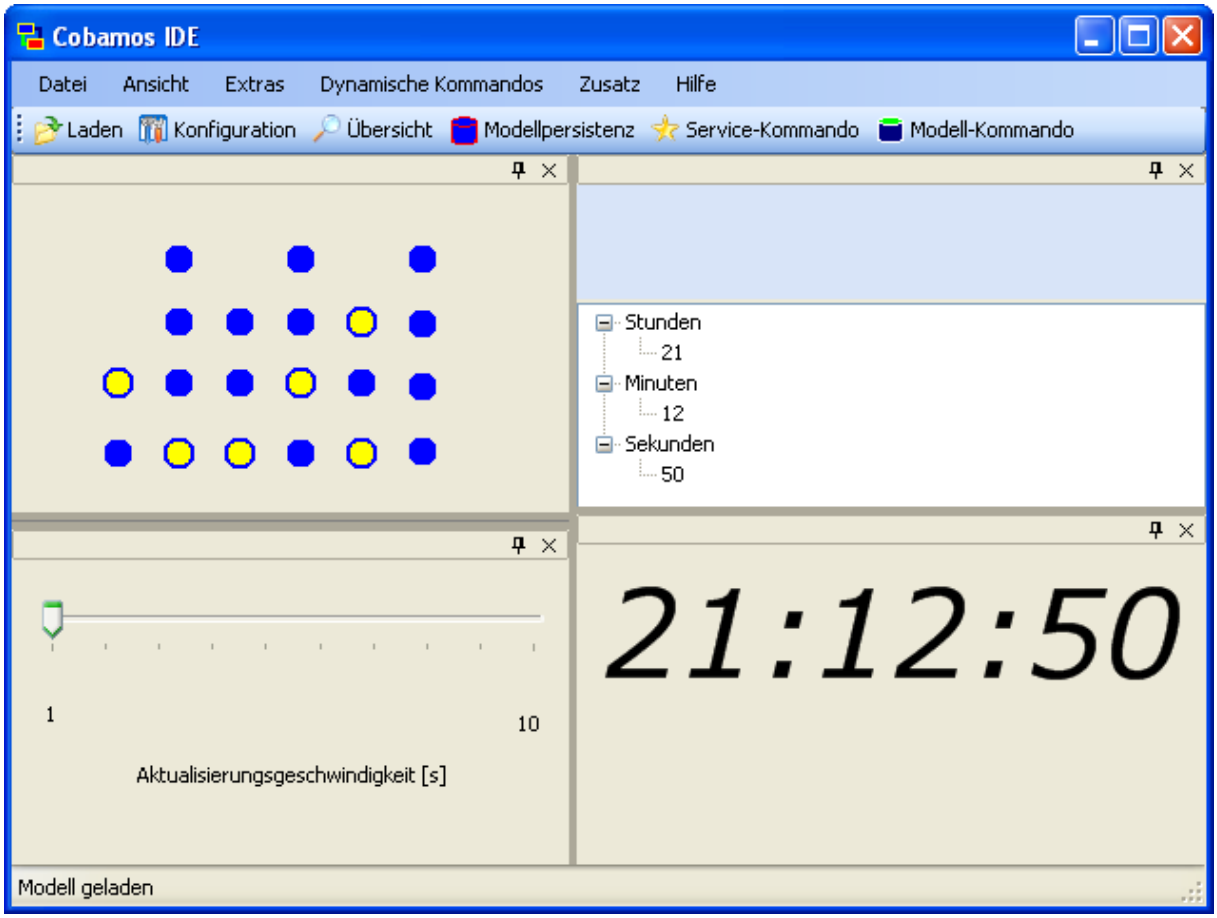


Abb. 6.3: Die Cobamos-IDE mit geladenem „Uhrmodell“

In Abb. 6.3 ist die Cobamos-IDE mit geladenem Modell, der oben angesprochenen Zeitanzeige, abgebildet. Bei allen vier Fenstern, die in der IDE geöffnet sind, handelt es sich um Sichten auf das Modell. Drei davon zeigen auf unterschiedliche Art und Weise die Uhrzeit[60] an, die vierte stellt nur einen Schieberegler dar. Über diesen kann die Aktualisierungsgeschwindigkeit, mit der das Modell die Sichten über Zeitänderungen informiert, eingestellt werden. Das Zeitintervall reicht dabei von einer bis zu zehn Sekunden. Diese Sicht wird zwar auch über Änderungen am Modellzustand informiert, nutzt dies aber nur um den angezeigten Intervallwert an den des Modells anzupassen, da es prinzipiell denkbar wäre, dass dieser über eine weitere Sicht geändert wird. Damit ist diese Sicht auch ein Beispiel für eine Beschränkung auf einen Aspekt des Modells.

Diese Kombination besitzt alle Eigenschaften, die für einen Test der im Cobamos-Framework rund um das MVC'-Muster implementierten Funktionalitäten notwendig sind. Darüber hinaus verdeutlicht dieses Beispiel eine sekundäre Einsatzmöglichkeit des entworfenen IDE-Frameworks. Im Falle eines

[60] Auch die Ansammlung aus Punkten im Bereich links oben zeigt die Uhrzeit an, allerdings in einer ungewohnten Notation. Zur Codierung der Ziffern wird eine binäre Darstellung verwendet, je eine Punktsäule entspricht einer Ziffer der Uhrzeit. Die Potenzen sind von unten nach oben angeordnet, die unterste Zeile entspricht 2^0, die nächste entspricht 2^1 und so fort. Die gelben Punkte mit blauem Rand zeigen in dieser Binärdarstellung eine 1, die blauen ein 0 an. An jeder Stelle ist nur die zur Codierung der dort möglichen Ziffern nötige Anzahl von Punkten vorhanden.

ausführbaren Modells, wie es beispielsweise im Falle von Simulationen Einsatz findet, kann das Framework als Basis zur Implementation von Simulationsumgebungen verwendet werden. Derartige Simulationen spielen in vielen Arbeiten aus dem Bereich der Sportinformatik, wie sie am Institut für Informatik der Johannes Gutenberg-Universität durchgeführt wurden und werden, eine große Rolle. Durch die Simulation des Modellverhaltens können – insbesondere wenn der Nutzer ausreichende Interaktionsmöglichkeiten besitzt – auch komplexe Systeme verstanden werden. Gerade wenn das modellierte System selbst nicht oder nur schlecht zu beobachten ist, bietet die Simulation eines entsprechenden Modells eine Möglichkeit des Zugangs (siehe auch [PB97]).

Zur Realisierung des MVC'-Konzepts muss wie in 4.4.2 dargelegt das Observer-Muster realisiert werden. Dabei kommt aus den dort erörterten Gründen eine Schnittstelle und keine abstrakte Oberklasse für die Beobachter des Modells zum Einsatz. In C# bietet es sich an, Ereignisse und so genannte Delegaten zur Realisierung zu verwenden. Delegaten (Schlüsselwort `delegate`) sind, wie in Kapitel 5 bereits kurz erwähnt, im Grunde genommen eine objektorientierte Variante von Funktionszeigern. Diese werden in C# durch vollwertige Klassen realisiert (siehe [Mic06i]). Die Deklaration der Klasse legt die Parameter der Methoden und ihre Rückgabewerte fest; dieser Ansatz ermöglicht typsichere Referenzen auf Methoden. Eine Instanz der Delegatenklasse kann jedoch im Unterschied zu einem Funktionszeiger nicht nur eine Referenz auf eine einzige Funktion enthalten, sondern eine Menge von Referenzen auf Methoden mit der passenden Signatur. Ein Ereignis (Schlüsselwort `event`) in C# entspricht einem Delegaten, für den einige zusätzliche Bedingungen hinsichtlich der Zugriffsrechte gelten. Insbesondere kann ein Ereignis nur durch die Klasse ausgelöst werden, in der es angesiedelt ist. Das Auslösen des Ereignisses entspricht dem Aufruf aller Methoden, auf die zu diesem Zeitpunkt eine Referenz gespeichert ist. In der in 4.4.2 gezeigten Variante des Observer-Musters wird in der Subjektklasse – diese entspricht dem Modell des MVC'-Musters – eine Liste mit Referenzen auf die Beobachterinstanzen verwaltet. Im Falle einer Änderung wird diese Liste durchlaufen und für jede Instanz die Methode `Update()` aufgerufen. Durch die Verwendung von Delegaten und Events muss in der Subjekt-Klasse nur ein Ereignis vom Typ des zu der Update-Methode passenden Delegaten deklariert werden. Damit ein Beobachter über Änderungen am Subjekt informiert werden kann, muss zu diesem Ereignis eine Referenz auf eine entsprechende Instanzmethode der Beobachterklasse hinzugefügt werden. Im Falle einer Änderung wird in der Subjektklasse „das Ereignis aufgerufen“, dies bewirkt in der Folge den Aufruf aller referenzierten Delegaten.

Angewandt auf das MVC'-Muster bedeutet dies, dass dem Änderungsereignis des Modells beim Hinzufügen einer Sicht deren Behandlungsmethode für Modelländerungen hinzugefügt werden muss. Das folgende Codebeispiel 6.4 zeigt den Quelltext der Methode `AddView()` aus der Klasse `MVCManager` des Cobamos-Frameworks. Diese Klasse stellt die zur Verwaltung von Modell, Sichten und Connector notwendige Funktionalität bereit.

```
(1)   public void AddView(IModelObserver view)
(2)   {
(3)       _views.Add(view);
(4)       _model.Changed += new ModelChanged(view.OnModelChanged);
(5)       view.Unload += new EventHandler(this.OnObserverUnload);
(6)       if (view is IView)
(7)       {
(8)          ((IView)view).Model = _model;          //Hinzufügen Modell
(9)          ((IView)view).Connector = _connector;  //Hinzufügen Connector
(10)      }
(11)      OnMVCChanged(EventArgs.Empty);
(12)  }
```

Codebeispiel 6.4: Hinzufügen einer Sicht

Beim Entwurf des Frameworks wurde zwischen reinen Modellbeobachtern, die nur über Änderungen informiert werden sollen, und den eigentlichen Sichten unterschieden. Zu diesem Zweck wurden die Schnittstellen `IModelObserver` beziehungsweise `IView` eingeführt (vergleiche Abschnitt 4.4.2 auf Seite 73). Diese Unterscheidung schlägt sich in der Implemementation der Methode `AddView()` nieder. Sowohl Modellbeobachter als auch Sichten werden in die Liste der vorhandenen Beobachter eingefügt. Ihre Benachrichtigungsmethode wird dem Modell bekannt gemacht und dem Ereignis zum Entladen des Beobachters wird die entsprechende Methode der Klasse `MVCManager` hinzugefügt (Zeilen 3 bis 5). Handelt es sich um eine Sicht, so wird dieser zusätzlich eine Referenz auf das Modell und auf dem Connector übergeben (Zeilen 6 bis 11). Am Ende der Methode wird das Ereignis der Verwaltungsklasse ausgelöst, welches eine Änderung an den verwalteten MVC'-Bestandteilen signalisiert. Dieses wird beispielsweise verwendet, damit die in Abb. 4.18 auf Seite 77 gezeigte Übersicht immer den aktuellen Zustand im Hinblick auf geladenen Sichten, Modell und Connector darstellen kann.

6.5 Realisierte Dienste

Nachdem die im Rahmen des Cobamos-Frameworks realisierten Dienste in den vorangegangenen Abschnitten bereits einige Male Erwähnung fanden, soll hier eine kurze Gesamtdarstellung gegeben werden. Wie im Entwurf der serviceorientierten Architektur festgelegt, müssen alle Dienste die Schnittstelle `IService` realisieren, welche die Methoden deklariert, die zur Verwaltung der Dienste innerhalb des Frameworks notwendig sind.

Bei den Diensten lassen sich grundsätzlich zwei Gruppen unterscheiden. Zum einen die Dienste, die mit hoher Wahrscheinlichkeit im Laufe der weiteren Evolution des Frameworks oder bei Anpassungen auf konkrete Einsatzszenarien Veränderungen erfahren werden. Zum anderen jene Dienste, die für das Funktionieren der IDE eine sehr zentrale Rolle spielen und von vielen anderen Diensten benötigt werden, bei denen eine Anpassung aber derzeit unwahrscheinlich erscheint, oder bei denen eine Anpassung tiefer in das Framework eingreift. In der Realisierung schlägt sich diese Unterscheidung in der Form nieder, dass für die erste Gruppe der Dienste die Schnittstellen zum Zugriff auf die bereitgestellte Funktionalität als separate Interfaces entworfen wurden. Dies führt zu einer loseren Kopplung dieser Dienste an das Framework. Die Interfaces der Dienste stehen in getrennten Assemblys zur Verfügung. Auf dieser Basis kann auch von Entwicklern ohne Zugriff auf die Interna des Frameworks eine angepasste Version eines Dienst umgesetzt werden. Die zweite Gruppe ist direkt von der Klasse `AbstractService` abgeleitet, welche eine abstrakte Basisimplementation der Schnittstelle `IService` bereitstellt. Es ist prinzipiell möglich, diese Dienste ebenfalls zu ersetzen. Sie sind in der derzeitigen Realisierung in gleicher Weise in einer Komponente gekapselt und werden

beim Anwendungsstart dynamisch geladen, wie die Dienste der ersten Gruppe. Eine Anpassung erfordert aber genauere Kenntnis des Frameworks.

Das Einführen einer abstrakten Klasse zur Realisierung einer Schnittstelle in die Vererbungshierarchie hat sich während der Entwicklung des Cobamos-Frameworks als hilfreiche Entwurfstechnik bewährt. Auch bei sorgfältigem Entwurf lassen sich im Verlauf eines größeren Projektes Änderungen an den Schnittstellen – in diesem Fall zum Beispiel denen der Dienste – selten völlig vermeiden. Solche Änderungen lassen sich in der abstrakten Klasse durch geeignete Standardimplementationen zunächst abfangen. In der Folge bleibt die Anwendung auch bei einer Schnittstellenänderung lauffähig und die benötigten Änderungen können sukzessive umgesetzt werden. Während der gesamten Änderungsphase steht ein testbares beziehungsweise nutzbares System zur Verfügung. Diese Entwurfstechnik unterstützt somit eine iterative und inkrementelle Entwicklung der Gesamtanwendung. Als möglicher Nachteil muss die Gefahr angesehen werden, in einem undisziplinierten oder unstrukturierten Entwicklungsprozess unter Umständen notwendige Anpassungen in den Klassen zu übersehen, die das geänderte Interface implementieren. Ohne das Einführen der abstrakten Klasse würde der Compiler beim Übersetzungsprozess an den entsprechenden Stellen Fehlermeldungen ausgeben. An dieser Stelle ist der Hinweis angebracht, dass bei der Realisierung der Frameworkfunktionalität jedoch immer die Schnittstellen oder die konkreten Realisierungen als Typen verwendet werden sollten. Anderenfalls ergeben sich die in Kapitel 4 im Zusammenhang mit dem Entwurf von Frameworks und der Anpassung durch Vererbung angesprochenen Probleme. Die abstrakten Basisklassen stellen lediglich eine Erleichterung für den Entwickler dar.

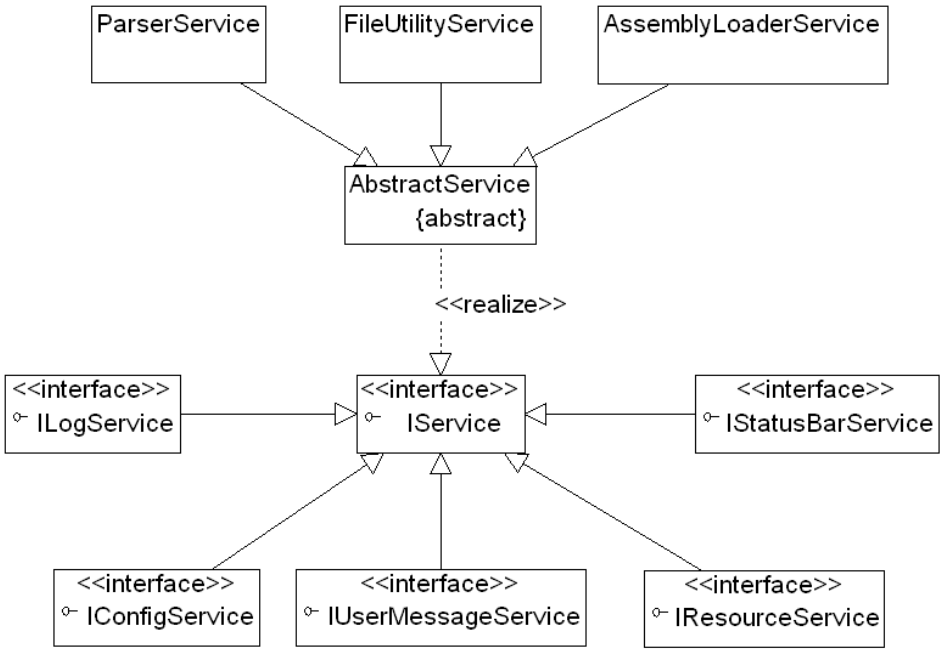


Abb. 6.4: Überblick über die realisierten Dienste im Cobamos-Framework

Das Klassendiagramm in Abb. 6.4 zeigt einen Ausschnitt des aus den obigen Überlegungen resultierenden Dienstesystems des Cobamos-Frameworks. Die Dienste `ParserService`, `FileUtilityService` und `AssemblyLoaderService` wurden ohne eigens deklarierte Schnittstellen umgesetzt[61]. Sie stellen Funktionalität zur Zeichenkettenverarbeitung, für Dateioperationen und zum

[61] Die Implementation der Dienste ParserService und FileUtilityService erfolgte unter Verwendung von modifiziertem Quellcode aus den Quellen zu [HKS04].

Umgang mit Assemblys bereit. Die von der Schnittstelle `IService` abgeleiteten Schnittstellen gehören (gegen den Uhrzeigersinn) zu den Diensten für die Ereignisprotokollierung, die Verwaltung von Konfigurationsdaten in externen Dateien, die Kommunikation mit dem Anwender durch Meldungsfenster, das Bereitstellen von Ressourcen – wie zum Beispiel Grafiken und Icons – und zum Zugriff auf die Statuszeile des Hauptfensters. Für alle diese Dienste wurde mindestens eine Realisierung im Rahmen der prototypischen Implementation umgesetzt.

Eine besondere Rolle kommt dem Dienst zu, der einen Zugriff auf das Modell (im Sinne des Modells als Bestandteil des MVC'-Musters) ermöglicht. Es gibt eine Reihe von Aufgaben, die sinnvollerweise unter Verwendung der dynamisch ladbaren Interaktionskomponenten realisiert werden, für die ein Zugriff auf das Modell notwendig ist. Hierzu zählt beispielsweise das Speichern und Wiederherstellten von Modellzuständen. Diese Funktionalität ist insbesondere dann notwendig, wenn auf Basis des Modells Anwendungen erstellt werden sollen. Müsste jede Entwicklung von Grund auf erfolgen, wäre dies ein erheblicher Nachteil. Unterschiedliche Modelltypen werden jedoch in der Regel angepasste Lösungen für die Serialisierung erfordern, die hierfür notwendige Funktionalität sollte also anpassbar sein.

Eine Lösung zum Zugriff auf das Modell sollte jedoch nicht auf das Szenario des Speicherns und Wiederherstellens von Modellzuständen beschränkt sein. Mit den Diensten existiert bereits ein Teilsystem des Cobamos-Frameworks, welches frameworkweit Funktionalität bereitstellt. Es liegt nahe, einen Dienst zu realisieren, welcher den Zugriff auf das Modell ermöglicht. Der Dienst ist aufgrund der realisierten Anwendungsarchitektur austauschbar und kann daher bei unterschiedlichen Modellen leicht angepasst werden, falls dies nötig sein sollte.

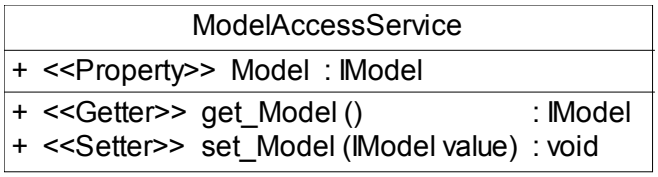


Abb. 6.5: Der Dienst zum Zugriff auf das Modell

In Abb. 6.5 ist der zur Realisierung eines einfachen Modellzugriffs entworfene Dienst dargestellt. Damit der Dienst Zugriff auf das Modell erlangen kann, wird er von der Klasse, welche die MVC'-Verwaltung realisiert, initialisiert und im Servicemanager eingetragen. Diese übergibt dem Dienst eine Referenz auf das Modell und trägt dafür Sorge, dass im Falle eines Wechsels des Programmiermodells auch der Dienst aktualisiert wird.

Die Klasse `ModelAccessService` ist von der abstrakten Klasse `AbstractService` abgeleitet und erbt von dieser die Standardimplementationen für die von den Diensten des im Cobamos-Frameworks zu implementierenden Verwaltungsmethoden. Dies ist das einfachste Vorgehen zur Realisierung eines neuen Dienstes im Cobamos-Framework. Falls eine besondere Funktionalität zum Initialisieren oder Beenden eines Dienstes benötigt wird, muss die Dienstklasse entweder direkt die Schnittstelle `IService` implementieren oder die entsprechenden Methoden der Klasse `AbstractService` überschreiben.

6.6 Realisierung eines dynamischen Kommandos am Beispiel Modellpersistenz

Auf der Basis des Dienstes für den Modellzugriff wurde – um der oben angesprochenen Notwendigkeit der Serialisierung von Modellzuständen Rechnung zu tragen – ein Persistenzkommando für Modelldaten realisiert. Anhand dieses Kommandos soll in diesem Abschnitt kurz das allgemeine Vorgehen zur Erstellung solcher Erweiterungen für die Entwicklungsumgebung aufgezeigt werden.

Die Anforderungen bestehen zum einen aus der Implementation der zur Integration in die Entwicklungsumgebung notwendigen Schnittstellen und zum anderen aus der eigentlichen Funktionalität zum Speichern und Wiederherstellen von Modellzuständen. Die Integration in die IDE besteht aus zwei Teilanforderungen. Die Implementation muss als dynamisch ladbare Kommandokomponente in der IDE nutzbar sein, daher muss sie das Interface `ICobamosCommand` (siehe Abb. 4.7 auf Seite 61) realisieren. Darüber hinaus benötigt sie Zugriff auf das Modell des MVC'-Musters. Wie oben erläutert, ist hierzu ein Zugriff auf den Dienst `ModelAccessService` notwendig. Die Realisierung der Kommandokomponente muss also auch die Schnittstelle `IServiceConsumer` implementieren.

Auch für die Schnittstelle `ICobamosCommand` steht innerhalb des Frameworks eine abstrakte Klasse mit einer Standardimplementation zur Verfügung. Diese findet auch bei der Realisierung des Persistenzkommnados Verwendung. In Abb. 6.6 sind die bisherigen Überlegungen dargestellt. Zur Integration in die visuelle Oberfläche der Entwicklungsumgebung müssen der sichtbaren Bezeichner, die Position im Menü und die Angabe, ob das Kommando auf der Werkzeugleiste angezeigt werden soll oder nicht bereitgestellt werden. Auch das optionale Icon wird in diesem Fall angegeben. Das Bereitstellen dieser Informationen geschieht durch Überschreiben der entsprechenden Eigenschaften der Schnittstelle `ICobamosCommand`. Die umzusetzende Funktionalität beschränkt sich hierbei auf ein einfaches Zurückliefern der jeweiligen Werte.

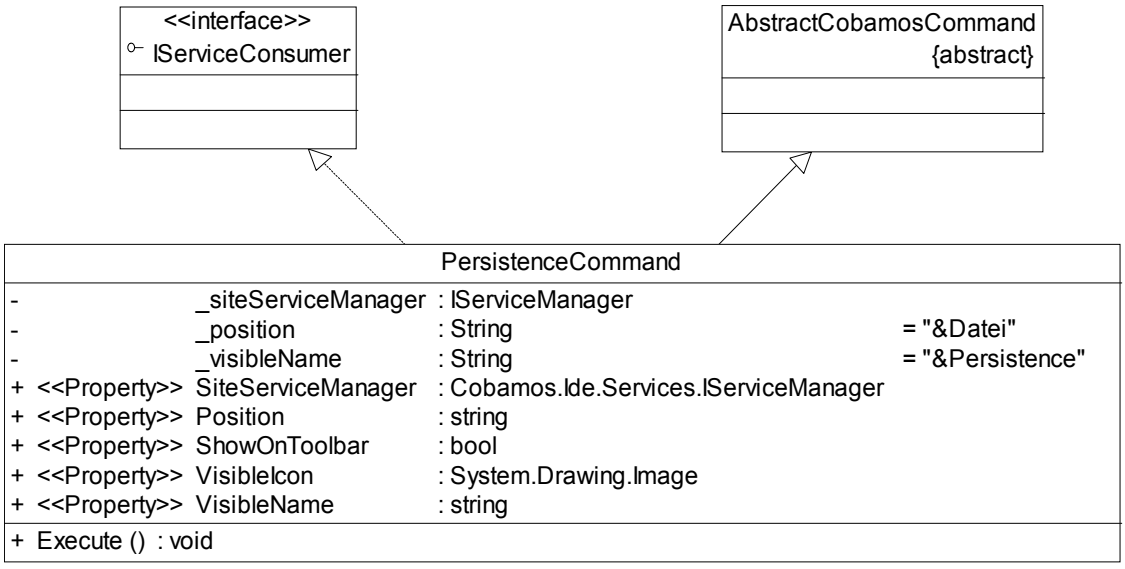


Abb. 6.6: Die Klasse PersistenceCommand

Die Methode `Execute()` (Codebeispiel 6.5) der Klasse `PersistenceCommand` wird bei einer Aktivierung des Kommandos über die Nutzungsoberfläche aufgerufen. Diese Methode muss also die eigentliche Funktionalität der Kommandokomponente beinhalten. Da für die geplante Funktionalität weitere Interaktionen mit dem Anwender notwendig sind, erzeugt die Methode `Execute()` eine Instanz vom

Typ `frmPersitence`. Diese Klasse implementiert ein Dialogfenster und benötigt eine Referenz auf den Dienst zum Zugriff auf das Modell, welche dem Konstruktor übergeben wird (siehe Zeile 4 und 5). Das modale Anzeigen (Zeile 6) delegiert den Kontrollfluss an das neu erzeugte Fenster, die Methode `Execute()` wird erst verlassen, wenn dieses Fenster wieder geschlossen wurde.

```
(1)   public override void Execute()
(2)   {
(3)      frmPersistence frm =
(4)        new frmPersistence( (ModelAccessService)_siteServiceManager.GetService(
(5)             typeof(ModelAccessService)));
(6)        frm.ShowDialog();
(7)   }
```

Codebeispiel 6.5: Die Methode Execute des Persistenzkommandos

Die durch die Klasse `frmPersistence` bereitgestellte Interaktionsoberfläche des Persistenzkommandos ist in Abb. 6.7 gezeigt. Wie bereits angedeutet, ist die eigentliche Funktionalität zur Speicherung in dieser Klasse realisiert. Die Daten des Modells werden in einer objektorientierten Datenbank abgelegt, da dieser Ansatz die – nicht immer unproblematische, siehe zum Beispiel [CB05] – Abbildung objektorientierter Strukturen auf eine relationale Datenbank vermeidet.

Anwendung findet hier die objektorientierte Datenbank db4o der Firma db4objects [db406]. Diese Datenbank lässt sich innerhalb des .NET-Frameworks direkt verwenden. Darüber hinaus muss die Datenbank nicht installiert werden, die benötigte Bibliothek ist nur[62] etwas mehr als 800 KB groß. Die Größe der Bibliothek ist im Rahmen des hier beschriebenen Einsatzes ein Kriterium, da sie im Verhältnis zum Nutzen und der Gesamtgröße des Persistenzkommandos stehen muss. Ebenso wäre ein Datenbanksystem, welches eine umfangreiche Installation und Konfiguration erfordert, nicht für die Verteilung mit der Komponente geeignet.

[62] Diese Angabe bezieht sich auf die verwendete Version 5.2 für .NET.

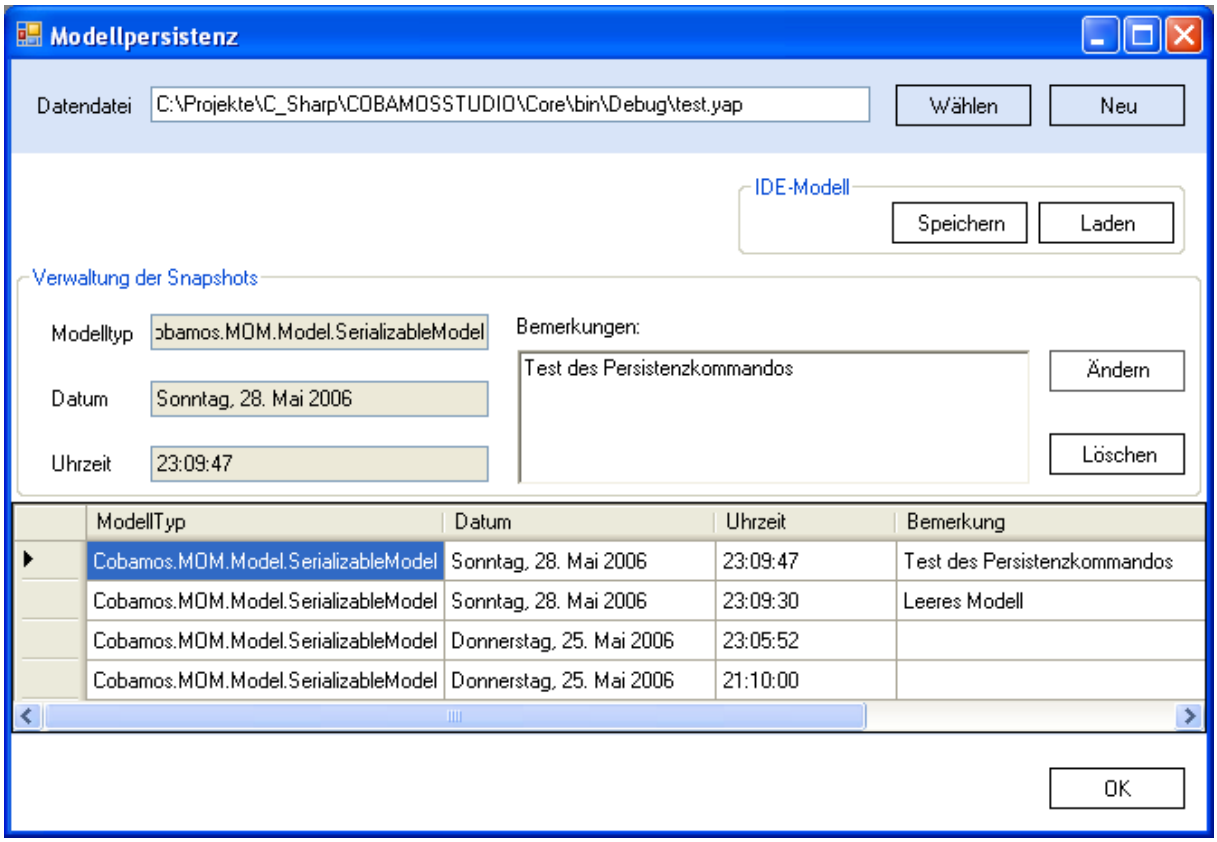


Abb. 6.7: Die Interaktionsoberfläche des Persistenzkommandos

Die Interaktionsoberfläche des Persistenzkommandos lässt sich anhand der notwendigen Prozessschritte in drei Teile untergliedern. Im oberen, hellblau eingefärbten Teil des Fensters kann eine Speicherdatei ausgewählt oder angelegt werden. In dieser Datei werden die Modelldaten gespeichert. Das Speicherkonzept ist dabei so angelegt, dass in einer Datei mehrere Modellzustände – quasi als Schnappschüsse – abgelegt werden können. Im unteren Bereich des Fensters werden alle in der aktuell ausgewählten Datei vorhandenen Modellzustände angezeigt. Datum, Uhrzeit und Typ des Modells für einen solchen Schnappschuss werden automatisch vergeben, der Bemerkungstext kann frei eingegeben und auch noch später geändert werden. Die Schaltflächen in der Gruppierungsbox „IDE-Modell“ dienen zum Übertragen („Laden“) eines selektierten Modellzustandes in das aktuell in der Entwicklungsumgebung vorhandene Modell beziehungsweise zum Anlegen („Speichern“) eines neuen Schnappschusses mit dem aktuellen Modellzustand der IDE.

Die Modelldaten werden nicht direkt in der objektorientierten Datenbank gespeichert, sondern zusammen mit den Metadaten (Datum, Uhrzeit, Typ, Beschreibung) in einem Objekt gekapselt. Auch beim Einsatz mit unterschiedlichen Modelltypen sind daher die in der Datenbank zu behandelnden Instanzen immer vom gleichen Typ.

Anhand des Modelltyps kann beim Laden von Modelldaten überprüft werden, ob die Daten eines Schnappschusses zum aktuellen Modell der Entwicklungsumgebung passen. Dies ist nötig, da die Entwicklungsumgebung ja entworfen wurde, um unterschiedliche Programmiermodelle zu unterstützen und das Persistenzkommando in möglichst vielen Fällen Anwendung finden soll.

Beim Speichern der Modelldaten muss berücksichtigt werden, dass bestimmte Typen, die in einer Modellinstanz Anwendung finden, unter Umständen nicht in unveränderter Form in der Datenbank gespeichert werden können. Dies betrifft beispielsweise Delegateninstanzen. Daher kann eine Instanz der Modellrealisierung zur Speicherung nicht direkt verwendet werden.

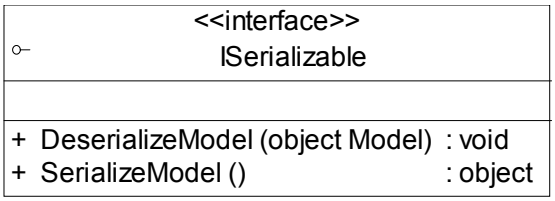


Abb. 6.8: Die Schnittstelle ISerializable

Die Verantwortlichkeit, eine serialisierbare Form der Modelldaten zu erzeugen beziehungsweise einen Modellzustand aus der serialisierten Form wiederherzustellen, wird sinnvollerweise bei der Modellrealisierung angesiedelt. Würde stattdessen eine Adapterklasse Verwendung finden, müsste diese aufgrund der Austauschbarkeit der Modelle bei einem Modellwechsel immer mitgetauscht werden. Diese direkte Kopplung kann nicht vermieden werden. Zur Umsetzung wird die in Abb. 6.8 gezeigte Schnittstelle `ISerializable` eingeführt. Diese legt die notwendigen Methoden zum Überführen des Modellzustandes in eine serialisierbare Form und zum Deserialisieren fest. Um möglichst geringe Einschränkungen hinsichtlich des Vorgehens zur Serialisierung zu machen, findet die Klasse `object` als Datentyp für die serialisierten Modelldaten Anwendung. Da diese Klasse die Basisklasse aller Klassen in C# ist, können hier beliebige Datenstrukturen Anwendung finden.

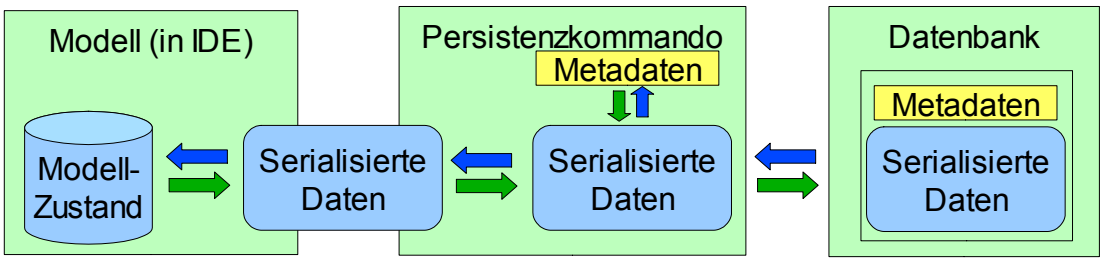


Abb. 6.9: Datenfluss bei der (De-)Serialisierung eines Modellzustandes

Der eben beschriebene Prozess, der zur Speicherung der Modelldaten und zum Laden eingesetzt wird, ist zusammenfassend anhand des Datenflusses in Abb. 6.9 dargestellt. Die grünen Rechtecke symbolisieren in der Grafik die jeweils verantwortlichen Teilsysteme.

Analog zum Zugriff auf den Dienst, welcher den Modellzugriff bereitstellt, kann – wie in Kapitel 4 bereits erwähnt – von einer dynamischen Interaktionskomponente auch ein Zugriff auf einen beliebigen anderen Dienst erfolgen. Dies kann ausgenutzt werden, um Nutzungsoberflächen für Dienste zu realisieren. Ein Einsatzszenario hierfür ist die Konfigurationen der Dienste durch einen Anwender. Wird ein Dienst ausgetauscht oder sollen die Anwender in einem Einsatzszenario der Entwicklungsumgebung keinen Zugriff auf die Konfigurationsmöglichkeiten eines Dienstes haben, so kann die Komponente, die das zugehörige Kommando realisiert, einfach ausgetauscht oder entfernt werden.

6.7 Zusammenfassung und Ausblick

Dieses Kapitel stellt einige Aspekte der Umsetzung und Anwendung des zuvor entworfenen Frameworks unter Verwendung der Sprache C# dar. Der Schwerpunkt liegt bei den Umsetzungsdetails auf den Bereichen der Realisierung, die zum einen allgemein genug sind, um in anderen

Zusammenhängen von Interesse zu sein und zum anderen eine Bedeutung bei der Implementation komponentenbasierter Lösungen besitzen.

Darüber hinaus verdeutlichen die in diesem Kapitel dargelegten Beispiele der prototypischen Realisierung die Tragfähigkeit der in Kapitel 4 ausgeführten Überlegungen und Entwurfsentscheidungen. Neben den gezeigten Eigenschaften der Entwicklungsumgebung ist noch erwähnenswert, dass die gesamte Oberfläche der Anwendung unter Verwendung von austauschbaren Zeichenketten realisiert ist. Dies ermöglicht eine leichte Anpassung an andere Einsatzszenarien und Anwender.

In den folgenden Kapiteln werden nun Anwendungen auf Basis des Cobamos-Frameworks dargestellt. Dabei handelt es sich nach den konzeptuellen Grundlagen und den Abhandlungen zum Cobamos-Framework um den dritten Teilbereich dieser Arbeit.

7 Nachrichtenbasierte Komponentenkomposition

„Panta rhei, ouden menei.“
(Alles fließt, nichts bleibt.)
Heraklit, griechischer Philosoph, ca. 535 - 475 v. Chr.

In diesem Kapitel werden die konzeptionellen Grundlagen für ein Programmiermodell dargelegt, welches in den anschließenden Kapiteln auf Basis des Cobamos-Frameworks realisiert wird. Dieses Programmiermodell basiert auf der Verwendung von Nachrichten zur Komposition von Komponenten und ist für ein breites Spektrum von Anwendungen geeignet.

Nach einer Motivation der nachrichtenbasierten Kommunikation werden zunächst einige für den weiteren Verlauf der Überlegungen notwendige Begriffe definiert. Im Anschluss wird die grundlegende Idee des Kompositionsmodells dargestellt, auch auf den Punkt der für die Nachrichten verwendeten Sprachen wird an dieser Stelle kurz eingegangen. Nach einer Betrachtung denkbarer Kommunikationsszenarien zwischen Komponenten wird auf einer hohen Abstraktionsebene ein Entwurf für eine Anwendung vorgestellt, mit der diese Szenarien umgesetzt werden können. Das Kapitel schließt mit der Darstellung einiger Vorteile, weiterer Möglichkeiten und Nachteile des nachrichtenbasierten Ansatzes.

7.1 Einleitung und Motivation

Nachdem in den vorangegangenen Kapiteln dieser Arbeit das Cobamos-Framework als Basis zur Realisierung von Programmierumgebungen für Endnutzer beschrieben wurde, soll im verbleibenden Teil dieser Dissertation eine Anwendung der bisherigen Ergebnisse gezeigt werden. Der Begriff des Programmiermodells und der Zusammenhang mit der Entwicklungsumgebung wurde in Abschnitt 4.4 diskutiert, in diesem Kapitel werden die konzeptuellen Grundlagen für das Programmiermodell gelegt, dessen Umsetzung in Kapitel 8 erläutert wird. Der angedachte Einsatzbereich dieses Programmiermodells ist die Realisierung einer Softwareapplikation durch einen Endnutzer.

Um Endanwender in die Lage zu versetzen, eine Softwareapplikation zu entwickeln, ist im Vergleich mit der Softwareerstellung durch professionelle Entwickler ein Anheben der Abstraktionsebene, auf der die Software beschrieben wird, sinnvoll. Bei dem in dieser Arbeit beschriebenen Ansatz wird dies durch die Verwendung von Komponenten als Basis für die Anwendung erreicht. Die Granularität dieser Bausteine bestimmt maßgeblich die Abstraktionsebene, auf der sich der Benutzer bei der Beschreibung einer Softwarelösung bewegen muss.

Das Konzept der Komponenten erlaubt den Einsatz von – im Vergleich mit anderen Wiederverwendungskonzepten recht großen – Anwendungsbausteinen. Dadurch wird die zur Realisierung einer Applikation notwendige Beziehungskomplexität geringer. Allerdings geht dieser Vorteil zwangsläufig mit einem Verlust an Flexibilität einher. Diese Einschränkung ist jedoch im Hinblick auf die geplante Verwendung der Ergebnisse für die Endnutzerprogrammierung und die erreichbare Verringerung der Komplexität akzeptabel.

Der Aufbau einer Applikation aus einzelnen Komponenten erfordert ein Konzept zu deren Komposition. Im Folgenden wird hierfür ein Ansatz vorgestellt, welcher auf der Verwendung von Nachrichten basiert. Die Darstellungen des Programmiermodells bewegen sich auf einer konzeptionellen Ebene und vermeiden zunächst die Einschränkungen auf ein konkretes Anwendungsszenario. Entsprechende Anwendungen und Einsatzmöglichkeiten der Ergebnisse werden im Kapitel 9 vorgestellt, um die Tragfähigkeit des Konzeptes zu zeigen.

Zu den konzeptuellen Betrachtungen in diesem Kapitel ist an dieser Stelle anzumerken, dass sie sich auf die für den weiteren Verlauf der Arbeit notwendigen Aspekte des nachrichtenorientierten Ansatzes konzentrieren. Die Kommunikation zwischen Entitäten auf Basis von Nachrichten ist ein sehr weites Feld innerhalb der Informatik, die Anwendungen reichen von Computernetzen bis zu autonomen Agenten. Selbst bei einer Beschränkung auf die Kommunikation zwischen Komponenten ist eine vollständige Abhandlung des Themas an dieser Stelle nicht zu leisten. An einigen Stellen werden Ausblicke gegeben, die über die in dieser Arbeit benötigten bzw. umgesetzten Ansätze hinausgehen, dies wird im Text entsprechend gekennzeichnet. An anderen Stellen muss eine Beschränkung auf einfachere Realisierungsmöglichkeiten erfolgen, da eine allgemeine Lösung über den Rahmen dieser Dissertation hinausginge.

7.2 Nachrichtenbasierte Komposition von Komponenten

7.2.1 Einleitung

Die Kommunikation zwischen verschiedenen Entitäten innerhalb einer Softwareanwendung durch Nachrichten zu beschreiben, ist seit den Anfängen der Objektorientierung eine zentrale Idee dieses Paradigmas (siehe z. B. [BC89] [Weg90] [Ber93]). Dabei wird von der in der strukturierten Programmierung verwendeten Sichtweise eines Prozeduraufrufes abstrahiert. Eine Methode eines Objektes wird nicht aufgerufen, vielmehr bekommt das Objekt eine Nachricht gesendet, mit welcher es aufgefordert wird, diese Methode auszuführen. Das Objekt reagiert wenn möglich mit dem Ausführen der passenden Methode. Diese Betrachtungsweise betont die Eigenständigkeit der Objekte und die Kapselung der Methoden innerhalb der Objekte. Das Verhalten für den Fall, dass ein Objekt nicht über die passende Methode verfügt, ist nicht einheitlich festgelegt. Auf diesen Punkt wird weiter unten genauer eingegangen.

In einigen Arbeiten aus dem Bereich der Objektorientierung (siehe z. B. [ZCZ97]) wird eine Realisierung der Nachrichten als eigenständiges[63] Sprachkonzept gefordert. Die konsequente Weiterführung dieses Gedankens führt zum Paradigma des Nachrichtenorientierten Programmierens (Message Oriented Programming, MOP); in diesem Ansatz werden Nachrichten als so genannte „first class“ Konzepte für Programmiersprachen mit eigenen Metaklassen gesehen. Dieser Ansatz ist laut [Tho04] noch nicht sehr weitgehend untersucht.

Im Zusammenhang mit nachrichtenorientierter Programmierung ist eine Arbeit von Wall [Wal82] zu erwähnen, in welcher die Betrachtung von – in diesem Fall hauptsächlich verteilten – Algorithmen aus der Perspektive einer Nachricht anstelle der üblichen prozessorientierten Sichtweise als vielfach einfacherer Zugang zum Verständnis beschrieben wird. Da vermutet werden kann, dass sich diese Beobachtung auf das Verständnis einer Applikation überträgt, wird der nachrichtenbasierte Ansatz in dieser Arbeit im Bereich der Endnutzerprogrammierung eingesetzt.

In den verbreiteten objektorientierten Programmiersprachen wird die angesprochene konzeptuelle Betrachtungsweise der Objektorientierung, die Kommunikation zwischen Objekten als auf Nachrichten basierend zu betrachten, in unterschiedlicher Form umgesetzt. Insbesondere die Frage, wie sich ein Objekt verhält, wenn es auf eine Nachricht nicht reagieren kann, ist in diesem Zusammenhang interessant. In der Programmiersprache C# werden Operationsaufrufe zum Beispiel nicht als Nachrichten an Objekte realisiert, zumindest sind sie nicht in dieser Form in der Sprache zugänglich. Eine nichtinterpretierbare Nachricht an ein Objekt entspricht hier immer einem

[63] Im Sinne der Unabhängigkeit von Klassen- beziehungsweise Methodendefinitionen.

nichtausführbaren Methodenaufruf (die Nachricht adressiert eine nichtvorhandene Methode), je nach Art der Bindung führt dies zur Übersetzungs- oder zur Laufzeit zu einem Fehler. In der Skriptsprache Ruby [TFH05] [Rub06] hingegen wird das Nachrichtenkonzept in der Form umgesetzt, dass alle Operationsaufrufe Nachrichten an Objekte darstellen. Hieraus ergeben sich interessante Möglichkeiten für die Metaprogrammierung, insbesondere auch für den Fall, dass ein Objekt nicht über die Methode verfügt, welche durch eine Nachricht adressiert wird. So ist beispielsweise ein Umleiten von Nachrichten an andere Objekte oder auch ein Aufzeichnen und späteres Wiedergeben von Nachrichten möglich. Das folgende Beispiel (basierend auf [Wei06b]) verdeutlicht diese Möglichkeiten anhand der Implementation eines einfachen Proxys in Ruby. Interessant an dieser Realisierung ist, dass der Proxy unabhängig von einer konkreten Klasse ist. Er kann Nachrichten an Instanzen beliebiger Typen verarbeiten und weiterleiten.

```
(1)   class Proxy
(2)     def initialize
(3)       @messages = []
(4)     end
(5)     def method_missing(method, *args, &block)
(6)       @messages << [method,args,block]
(7)     end
(8)     def send_to(obj)
(9)       @messages.each do |method,args,block|
(10)        obj.send(method,*args,&block)
(11)      end
(12)    end
(13)  end
(14)
(15)    #Proxy-Instanz erzeugen und Nachrichten zwischenspeichern
(16)    proxy = Proxy.new
(17)    proxy.sub!(/little/){"Giant"}
(18)    proxy[12,5]="Universe"
(19)    proxy.upcase!
(20)    proxy<<"!"
(21)
(22)    #Anwenden auf einen String
(23)    string = "hello little world"
(24)    puts string #gibt “hello little world” aus
(25)    proxy.send_to(string)
(26)    puts string #gibt “HELLO GIANT UNIVERSE!” aus
```

Codebeispiel 7.1: Nachrichtenproxy in Ruby

Die Methode `method_missing()` wird in Ruby in der Klasse `Object` definiert, diese ist die gemeinsame Oberklasse aller Klassen in Ruby. Die Methode wird aufgerufen, wenn ein Objekt[64] nicht über eine geeignete Methode verfügt, um auf eine empfangene Nachricht zu reagieren. Um einen Proxy zu realisieren, wird diese Methode so überschrieben, dass die vom Proxy nicht zu verarbeitenden Nachrichten in einem Array gespeichert werden. Wird die Methode `send_to()` aufgerufen, so sendet diese die Nachrichten an das übergebene Objekt. In dieser Methode wird für die übergebene Instanz

[64] Im Zusammenhang mit Ruby muss an dieser Stelle in der Tat von Objekten gesprochen werden, da die Sprache es erlaubt, einzelnen Objekten zusätzliche Methoden hinzuzufügen. Diese sind dann nur in dieser konkreten Instanz verfügbar, nicht jedoch in allen Instanzen der zugehörigen Klasse. Dieses Konzept existiert noch in einigen anderen objektorientierten Sprachen (z. B. Dylan und JavaScript).

die Methode `send()` aufgerufen. Auch diese Methode ist in der Klasse `Object` deklariert und sendet die übergebene Aufrufnachricht an das Objekt.

Das im Beispielcode gezeigte Umleiten von Nachrichten an ein Objekt vom Typ String bewirkt dasselbe Verhalten, wie ein direktes Senden der entsprechenden Nachrichten an die Stringinstanz. Hier ist zu bemerken, dass keine Prüfung vorgenommen wird, ob das Zielobjekt die Nachrichten verarbeiten kann.

Nachrichtenbasierte Kommunikation eröffnet – auch wenn sie kein neuer Ansatz in der Softwareentwicklung ist – einige interessante Möglichkeiten, die im Folgenden näher betrachtet werden. Komponenten besitzen – wie in Kapitel 3 dargestellt – von der konzeptionellen Betrachtung her eine stärkere Eigenständigkeit als Objekte und sollen nur lose an eventuelle Nutzer ihrer Funktionalität und ihren Kontext gekoppelt sein. Darüber hinaus sind Komponenten auch für die Realisierung verteilter Systeme von Interesse, hier müssen jedoch geeignete Möglichkeiten der Kommunikation, insbesondere zwischen Komponenteninstanzen auf unterschiedlichen Rechnern, vorhanden sein. Aus diesen Gründen liegt es nahe, das Konzept der Nachrichtenorientierung aus der Objektorientierung auf Komponenten zu übertragen, um diese Ziele zu erreichen.

Bevor im Folgenden einige im Rahmen der Realisierung des Programmiermodells interessante Möglichkeiten eines nachrichtenbasierten Kompositionsmodells für Komponenten diskutiert werden, müssen einige grundlegende Begriffe definiert werden.

7.2.2 Begriffsbestimmungen und Definitionen

Dieser Abschnitt soll die – soweit diese einer Definition bedürfen – für die restlichen Ausführungen dieses Kapitels notwendigen Begrifflichkeiten definieren. Der grundlegende Begriff ist der der Nachricht. In den bisherigen Ausführungen wurde dieser Begriff bereits intuitiv verwendet, für den verbleibenden Teil dieser Ausarbeitung wird eine Nachricht wie folgt definiert.

> *Eine **Nachricht** (engl. Message) ist eine Menge von Daten, die von einem Sender entweder an einen spezifischen Empfänger oder an eine Menge von Empfängern gesandt wird. Es handelt sich dabei um ein diskretes Stück Information im Gegensatz zu einem Datenstrom.*

Nachrichten müssen nicht notwendigerweise zwischen verschiedenen Rechnern ausgetauscht werden. Sie können zwischen Anwendungen auf einem Rechner oder aber auch innerhalb einer Anwendung zwischen verschiedenen Komponenten ausgetauscht werden.

Es ist notwendig, zwischen einem Ereignis und einer Nachricht zu unterscheiden. Ein Ereignis (engl. Event) ist eine in Zeit und Raum beschränkte Begebenheit. Nachrichten sind eine Möglichkeit, ein Ereignis bekannt zu machen, dies wird beispielsweise bei der ereignisorientierten Programmierung für visuelle Oberflächen angewandt. Ein aufgetretenes Ereignis (z. B. Mausklick) wird hier durch entsprechende Nachrichten bekannt gemacht.

Im Sprachgebrauch und in der Literatur geht die Unterscheidung zwischen Ereignis und Nachricht teilweise verloren. Es werden sowohl das eingetretene Ereignis als auch die Nachricht, die es kommunizieren soll, als Ereignis bezeichnet. Um die Notwendigkeit der konzeptuellen Unterscheidung dieser Begriffe zu verdeutlichen, reicht es, sich bewusst zu machen, dass das Eintreffen einer Nachricht, welche ein eingetretenes Ereignis kommuniziert, selbst wieder ein Ereignis ist (siehe z. B. [SGM03]).

Die obige Definition sagt nichts über die Übertragung der Nachrichten aus. Der Übermittlungsmechanismus für Nachrichten zwischen zwei oder mehr Endpunkten wird im Folgenden als

Nachrichtenkanal bezeichnet. Im Moment werden noch keine Annahmen oder Einschränkungen hinsichtlich der Realisierung dieses Mechanismus gemacht. Insbesondere kann die Übermittlung sowohl synchron als auch asynchron erfolgen. Dies entspricht den existierenden Anwendungsbeispielen für die Verwendung von Nachrichten. Bei den oben angesprochenen Nachrichten zum Aufruf einer Methode im objektorientierten Paradigma handelt es sich in der Regel um eine synchrone Übermittlung[65], der Sender wartet auf eine Rückantwort des Empfängers. Die Verarbeitung von Nachrichten zur Übermittlung von Ereignissen bei visuellen Oberflächen erfolgt in der Regel asynchron, da ein Blockieren der Ereignisverarbeitung bis zum Ende der Behandlung eines aufgetretenen Ereignisses nicht wünschenswert ist.

7.2.3 Ein Kompositionsmodell für Komponenten auf Nachrichtenbasis

In diesem Abschnitt wird das eigentliche Programmiermodell, also das Konzept, auf dessen Basis später durch einen Endnutzer eine Softwareapplikation erstellt werden soll, dargelegt. Das hier vorgestellte Modell ist ein Metamodell. Es beschreibt auf einer abstrakten Ebene die Möglichkeiten zur Verknüpfung von Komponenten durch Nachrichten. Bei der späteren Anwendung im Rahmen der Endnutzerprogrammierung müssen konkrete Komponenten und Nachrichtenkanäle bereitgestellt werden, auf deren Basis ein Anwender Applikationen entwerfen kann. Die bereitgestellten Bausteine beschreiben dann eine Anwendungsfamilie, das die Anwendung beschreibende Modell ist eine Konkretisierung des hier entworfenen Metamodells.

Um ein Modell zur Komposition von Komponenten auf Nachrichtenbasis (im Folgenden auch Kompositionsmodell) realisieren zu können, muss zunächst eine Möglichkeit zum Nachrichtenaustausch zwischen den Komponenten oder genauer zwischen Komponenteninstanzen geschaffen werden. Betrachtet man Komponenten als Bausteine analog zu den integrierten Schaltkreisen der Elektronik (Integrated Circuit, IC), so liegt die Idee nahe, Konnektoren einzuführen, welche den Anschlussstellen der ICs entsprechen und Nachrichtenkanäle als Entsprechung der Leiterbahnen in das Modell aufzunehmen. Die Konnektoren – im Weiteren auch als Ports bezeichnet – lassen sich in zwei Gruppen unterscheiden. Zum einen solche Konnektoren, an denen Signale anliegen, und zum anderen die Konnektoren, die einen Signalausgang darstellen. Diese sehr anschauliche Herangehensweise führt – wie im Folgenden gezeigt wird – in der Tat zu einem tauglichen Modell für die nachrichtenbasierte Kommunikation zwischen Komponenteninstanzen.

Signalein- beziehungsweise -ausgänge entsprechen – übertragen auf das nachrichtenbasierte Kommunikationsmodell – dem Senden beziehungsweise Empfangen von Nachrichten. Die Bezeichnungen werden dabei aus Sicht der Komponente festgelegt, ein Eingangsport entspricht also dem Empfangen, ein Ausgangsport dem Senden von Nachrichten. Die Interaktion zweier Komponenten erfolgt dadurch, dass eine Komponente über einen ihrer Ausgangsports eine Nachricht an einen Eingangsport der anderen sendet. Das Eintreffen der Nachricht löst in der zweiten Komponente einen Verarbeitungsprozess aus. Dieser kann abhängig von den Daten sein, welche die Nachricht transportiert. Es sind aber auch Szenarien denkbar, in denen nur das Ereignis des Nachrichtenempfangs relevant ist. Darüber hinaus können auf dieser Basis sowohl Anwendungen realisiert werden, in denen die Komponenten fortwährend aktiv sind als auch solche, in denen nur durch Nachrichten Verarbeitungsprozesse angestoßen werden, welche nach einer gewissen Zeitspanne wieder enden[66].

[65] Die Möglichkeiten einiger Programmiersprachen (beispielsweise C#) zum asynchronen Methodenaufruf werden hier vernachlässigt.

[66] Bei den in Kapitel 9 vorgestellten Anwendungen existieren beide Varianten.

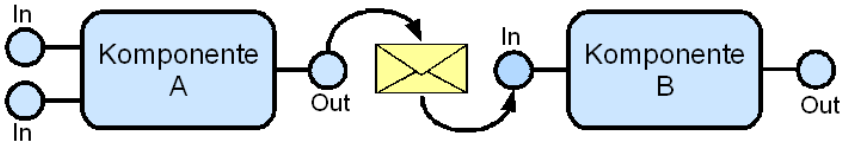


Abb. 7.1: Nachrichtenbasierte Komponentenkomposition

In Abb. 7.1 wird das Kompositionsmodell schematisch dargestellt. Eine Komponente verfügt über Ein- und Ausgangsports zum Empfangen respektive Senden von Nachrichten. Im hier dargestellten einfachsten Fall (siehe hierzu 7.3 Kommunikationsszenarien) werden die Nachrichten eines Ausgangs an einen Eingang einer anderen Komponente übertragen.

Das Modell erlaubt prinzipiell die Existenz mehrerer Ein- beziehungsweise Ausgänge einer Komponente. Im Falle mehrerer Eingänge gibt es zunächst kein zwingendes Verhalten bezüglich der Aktivierung der Verarbeitung in der Komponente. Es ist möglich, dass beim Eintreffen einer Nachricht an einem der Eingänge die Verarbeitung initiiert wird. Dies würde einem logischen „Oder“ entsprechen und ist beispielsweise beim Zusammenführen alternativer Pfade eines auf dieser Basis modellierten Prozesses notwendig. Aber auch ein Verhalten ähnlich einer Synchronisation bei Petri-Netzen (siehe z. B. [Per81] [RW82]) ist denkbar. Hier würde die Verarbeitung nur dann beginnen, wenn an jedem der Eingänge eine Nachricht eingetroffen ist. Dies entspricht einem logischen „Und“. Nötig ist dies beispielsweise, um eine Zusammenführung paralleler Pfade ausdrücken zu können. Bei der Anwendung dieses Modells muss an dieser Stelle eine Entwurfsentscheidung hinsichtlich des Verhaltens der Komponenten getroffen werden[67]. Existieren Komponenten mit unterschiedlichem Aktivierungsverhalten beim Eintreffen von Nachrichten an ihren Eingängen, so muss dies für einen Anwender nachvollziehbar dargestellt werden.

Der Fall, dass eine Komponente über mehrere Ausgänge verfügt, ist weniger interessant. Durch die lose Kopplung zwischen den Komponenten spielt es keine Rolle, wie viele der Ausgänge einer Komponente gleichzeitig eine Nachricht versenden. Eine sendende Komponente „weiß“ nicht, welche Anzahl von Nachrichten die empfangenden Komponenten benötigen.

Bisher wurde die grundsätzliche Idee des Programmiermodells motiviert und das Aktivierungsverhalten der Komponenten betrachtet. Im Zusammenhang mit einem Ansatz zur Komponentenkomposition muss auch das Problem der Kompatibilität der zu verbindenden Komponenten oder genauer der Konnektoren betrachtet werden. Dieses lässt sich in zwei Teilprobleme zerlegen. Eine Kompatibilität auf der syntaktischen Ebene entspricht einer Kompatibilität der Ein- beziehungsweise Ausgänge mit dem verwendeten Übermittlungsmechanismus, also dem Nachrichtenkanal. Diese syntaktische Kompatibilität des Übertragungsmechanismus kommt, bezogen auf das Beispiel der Schaltkreise, der Verträglichkeit der Steckverbindungen gleich. Durch den Einsatz von geeigneten Nachrichtenkanälen oder Adapterkomponenten sind prinzipiell Anpassungen syntaktischer Natur denkbar, allerdings tragen diese dazu bei, die Komplexität des Modells und der Realisierung zu erhöhen. Im Rahmen der späteren Anwendung stellt sich dieses Problem nicht, da alle realisierten Komponenten im Hinblick auf die Verwendung mit dem implementierten Übertragungsmechanismus entworfen wurden.

Wird der verwendete Übermittlungsmechanismus derart entworfen, dass er möglichst allgemein verwendbar ist, so tritt zwangsweise das Problem auf, dass keine Typsicherheit für die übermittelten Nachrichten gewährleistet ist. Typsicherheit bezieht sich hierbei zunächst auf den zur Realisierung der

[67] Die später im Rahmen der Anwendungsbeispiele in Kapitel 9 realisierten Komponenten mit mehreren Eingängen besitzen eine „Und“-Semantik.

Nachrichten verwendeten Typ (beispielsweise im Sinne einer Klasse im Falle einer objektorientierten Realisierung). Ein typsicherer Nachrichtenkanal würde gewährleisten, dass nur mit dem jeweiligen Typ konforme Nachrichten übertragen werden. Je allgemeiner der Nachrichtenkanal nutzbar ist, desto geringer sind die Einschränkungen, die bezüglich der übertragenen Nachrichten gemacht werden können. Die Typsicherheit eines Nachrichtenkanals kann jedoch über die eben angesprochene Betrachtung der zur Realisierung verwendeten Typen hinausgehend diskutiert werden. Lässt man diesen Typbegriff der Softwareentwicklung außer Acht, so kann der Typ einer Nachricht auch auf Basis ihrer Semantik festgelegt werden. Diese Art von Typsicherheit entspricht beispielsweise der Festlegung, dass ein Kanal nur Nachrichten weiterleitet, welche über Anwendungsfehler informieren.

Das Sicherstellen der semantischen Kompatibilität der Konnektoren stellt im Vergleich zur syntaktischen ein wesentlich größeres Problem dar. Wie bei allen Schnittstellenbeschreibungen erfordert eine vollständige Beschreibung der Semantik einen hohen Aufwand. Soll diese Beschreibung automatisiert verarbeitet werden, beispielsweise um als Basis für einen Kompositionsmechanismus oder zumindest für eine Unterstützung des Anwenders bei der Komposition zu dienen, so ergeben sich zusätzliche Anforderungen an ihre Genauigkeit. Im Folgenden wird ein Ansatz beschrieben, der für die Verwendung im Rahmen der weiteren Überlegungen und der Realisierung ausreichend ist. Die bei der Darstellung der Umsetzung des Programmiermodells in Kapitel 8 vorgestellte Methode zur Prüfung der Kompatibilität vom Komponentenkonnektoren beruht auf diesen Überlegungen. Eine allgemeine Lösung dieser Problemstellung kann im Umfang dieser Dissertation nicht erfolgen.

Neben der Semantik der Schnittstellen der Komponenten – dies würde einer Beschreibung der Verarbeitung der Nachricht oder der Reaktionen auf ihr Eintreffen entsprechen – kann zur Prüfung der Kompatibilität im hier diskutierten Modell jedoch auch das Vokabular der Nachrichten herangezogen werden. Als semantische Kompatibilität wird im Folgenden die Eigenschaft bezeichnet, dass der Empfänger die Nachrichten eines Senders korrekt interpretieren kann. Wie oben erläutert, wird diese Eigenschaft sinnvollerweise auf der Ebene von Konnektorpaaren definiert. Dass eine Nachricht von einem Empfänger interpretiert werden kann, setzt das Verstehen des verwendeten Vokabulars voraus. Dies bedeutet, dass zwei Konnektoren, damit sie in semantischer Hinsicht kompatibel sein können, zumindest über ein kompatibles Vokabular für die Nachrichten verfügen müssen. Um genauer zu sein, das Vokabular, welches der Empfänger versteht, muss eine Obermenge des Vokabulars (beziehungsweise der Sprache) sein, das dem Sender zur Verfügung steht. Diese Anforderung ist notwendig, jedoch nicht hinreichend. Dies wird am Beispiel der in Codebeispiel 7.2 gezeigten Nachricht deutlich. Das Vokabular der verwendeten Sprache basiert auf XML. Die abgebildete Nachricht dient offensichtlich zur Übermittlung eines ganzzahligen Wertes. Um die Nachricht verarbeiten zu können, muss ihr Vokabular – in diesem Fall `<message>`,`<value>` und `<integer>` – auf der empfangenden Seite interpretiert werden können. Trotz des gemeinsamen Vokabulars können immer noch „Missverständnisse“ auftreten; eventuell übermittelt die sendende Komponente in diesem Beispiel einen Zeitraum in Form der Anzahl der Tage, die empfangende Komponente erwartet jedoch die Anzahl von zu versendenden E-Mails.

```
(1)  <message>
(2)     <value>
(3)        <integer>42</integer>
(4)     </value>
(5)  </message>
```

Codebeispiel 7.2: Einfache Nachricht auf XML-Basis

Im Falle eines XML-basierten Nachrichtenvokabulars kann durch die Verwendung von Namensräumen (siehe [Wor01]) zumindest dafür Sorge getragen werden, dass identische Bezeichner in verschiedenen Sprachen unterschieden werden können.

Bezogen auf die spätere Anwendung dieser Überlegungen bei der Realisierung einer konkreten Programmierumgebung für Endnutzer ist anzumerken, dass die Überprüfung des notwendigen Kriteriums bereits ein wichtiger Schritt zur Unterstützung des Anwenders bei der Komposition von Komponenten ist. Durch diese Überprüfung auf die Kompatibilität der Nachrichtensprachen werden in jedem Fall solche Kompositionen verhindert, die auf keinen Fall korrekt sein können. Dieser Mechanismus kann mit einer kurzen Beschreibung der Funktionalität der Eingangs- beziehungsweise Ausgangsschnittstellen der Komponenten in natürlicher Sprache ergänzt werden. Diese kann beispielsweise beim Anlegen einer Verbindung angezeigt werden. Durch das Zusammenwirken dieser Mechanismen lässt sich mit einem vertretbaren Aufwand ein hohes Maß an Unterstützung des Endnutzers bei der Komposition einer Anwendung erreichen.

Im Zusammenhang mit dem obigen Beispiel ist es wichtig zu betonen, dass jede Nachricht – betrachtet als eigenständige Entität – in der Regel eine eigene, empfängerunabhängige Bedeutung besitzt, auch wenn die selbe Nachricht bei unterschiedlichen Empfängern ein abweichendes Verhalten bewirken kann (vgl. auch [ZCZ97]).

Bei den Überlegungen zu diesem Programmiermodell auf Basis von Nachrichten muss auch die Realisierung des Übermittlungsweges einbezogen werden, da dies für die modellierbaren Systeme nicht unerheblich ist. Bevor dessen Entwurf diskutiert wird, werden einige grundlegende Kommunikationsszenarien besprochen, welche unterstützt werden müssen.

7.3 Kommunikationsszenarien

Im Zusammenhang mit der oben vorgestellten Komponentenkomposition auf Basis von Nachrichten können unterschiedliche Übertragungsmechanismen mit unterschiedlichen Eigenschaften verwendet werden. Bei der Entwicklung des Programmiermodells sind diese mit entscheidend dafür, welche Anwendungssysteme modelliert werden können. Im weiteren Verlauf der Arbeit wird mit dem Ziel der Komposition von Komponenten ein Übermittlungsmechanismus für Nachrichten realisiert. Um die Anforderungen an diesen Mechanismus zu klären, werden zunächst einige grundlegende Kommunikationsszenarien zwischen Komponenten betrachtet, welche auf Basis des Übermittlungsmechanismus abbildbar sein sollen.

Je nach geplantem Einsatzszenario können auch unterschiedliche Variationen dieser Muster sinnvoll sein. Die folgende Darstellung beschränkt sich auf einen kurzen Überblick über die grundsätzlichen Möglichkeiten zur Realisierung einer nachrichtenbasierten Kommunikation zwischen Komponenten. Die Übersicht basiert zum Teil auf [W3C03] und [PBP+02].

Prinzipiell kann die Kommunikation zwischen den Komponenten, die das Anwendungssystem bilden, unter verschiedenen Gesichtspunkten betrachtet werden. Unterscheidungen sind zum Beispiel hinsichtlich der Anzahl der beteiligten Sender beziehungsweise Empfänger und hinsichtlich des erwarteten Antwortverhaltens möglich. Weitere Möglichkeiten der Differenzierung ergeben sich aus den Nachrichtentypen oder auch aus der Realisierung des zur Übermittlung verwendeten Kanals.

Zunächst werden die verschiedenen Möglichkeiten hinsichtlich der Anzahl der beteiligten Kommunikationspartner dargestellt. Die Fälle werden dabei unidirektional betrachtet, da eine eventuell vorhandene Kommunikation in die Gegenrichtung analog darzustellen wäre. Für jedes der nachfolgenden Kommunikationsmuster gilt gleichermaßen, dass eine konsumierte Nachricht aus dem

Kanal entfernt wird und dort nicht mehr zur Verfügung steht. Der Kanal wird zunächst nur als allgemeiner Übertragungsweg für Nachrichten betrachtet, ohne dass weitere Eigenschaften angenommen werden.

7.3.1 Direkte Kommunikation

Dieses Kommunikationsmuster (auch als **Punkt-zu-Punkt-Verbindung** bezeichnet) stellt den einfachsten Fall einer nachrichtenbasierten Kommunikation dar. Es sind genau zwei Kommunikationspartner vorhanden, einer in der Rolle des Senders, einer in der des Empfängers. Jede Nachricht wird vom Sender an den Nachrichtenkanal gesandt und der Empfänger konsumiert diese aus dem Nachrichtenkanal. Die unten stehende Abb. 7.2 verdeutlicht das Muster.

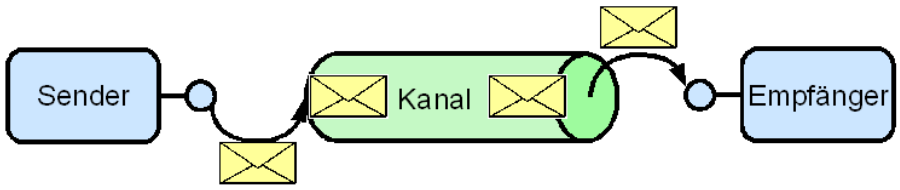


Abb. 7.2: Direkte Kommunikation

Jede Nachricht wird von genau[68] einem Empfänger konsumiert, da sie nach ihrer Entnahme aus der Warteschlange nicht mehr zur Verfügung steht. Diese Form der Kommunikation muss in jedem Fall mit dem zu entwerfenden Übermittlungsmechanismus realisierbar sein.

Eine Variation dieses Musters (und – entsprechend angepasst – auch der folgenden Kommunikationsmuster) besteht darin, dass ein Empfänger dem Sender den erfolgreichen Empfang einer Nachricht bestätigt. Dies kann auch im Falle eines asynchronen Übertragungskanals realisiert werden. Ein Szenario, in dem keine Bestätigung einer erfolgreichen Übertragung erforderlich ist, wird häufig als **fire-and-forget** (feuern und vergessen) bezeichnet. Der Sender fährt hierbei nach dem Verschicken der Nachricht mit seiner Verarbeitung fort, unabhängig von Erfolg oder Misserfolg der Nachrichtenübertragung.

7.3.2 Multicast-Kommunikation

Die zentrale Idee eines **Multicast** (oder einer **Mehrpunktverbindung**) ist die Kommunikation eines Senders mit mehreren Empfängern. Bei diesem Kommunikationsmuster müssen zwei unterschiedliche Ausprägungen betrachtet werden. Das Verhalten des Nachrichtenkanals ist derart definiert, dass eine Nachricht, wenn sie von einem Konsumenten empfangen wird, aus dem Kanal entfernt wird. Dies bedeutet, dass zwar durchaus mehrere Konsumenten auf einen Nachrichtenkanal zugreifen können, eine Nachricht aber immer nur einem Konsumenten zur Verfügung steht. Anwendung findet dieses Szenario beispielsweise, wenn mehrere gleichberechtigte Konsumenten für die Verarbeitung der Nachrichten bereit stehen. Dies kann zum Beispiel bei der Realisierung einer Lastverteilung der Fall sein. Die unten stehende Abb. 7.3 verdeutlicht dieses Kommunikationsmuster.

[68] Falls im betrachteten Szenario Nachrichten unzustellbar sein können oder verloren gehen können, muss es hier höchstens heißen.

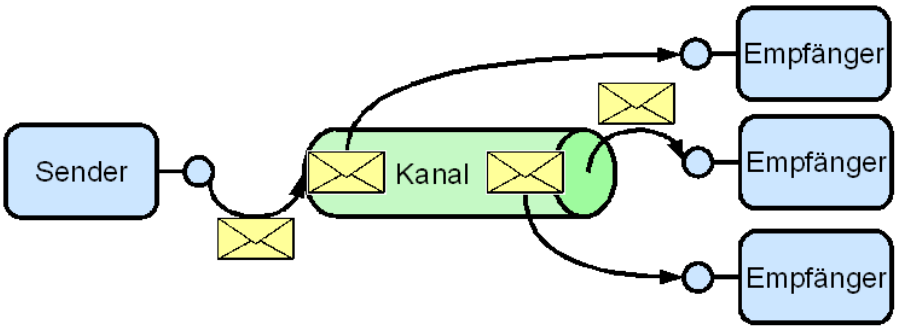


Abb. 7.3: Multicast-Kommunikation, serielle Variante

Die andere Variante dieses Musters ist in Abb. 7.4 dargestellt. Sie sieht vor, dass ein und dieselbe Nachricht tatsächlich von mehreren Konsumenten verarbeitet wird. Dies ist beispielsweise dann notwendig, wenn mehrere unabhängige Empfänger auf ein und dasselbe Ereignis reagieren oder Daten von mehreren Empfängern verarbeitet werden sollen.

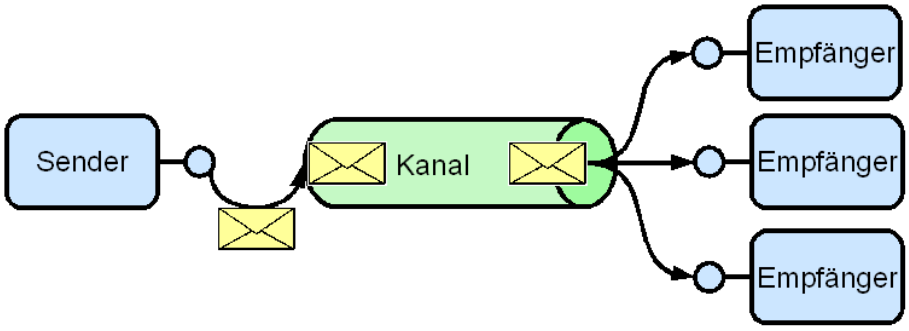


Abb. 7.4: Multicast-Kommunikation, parallele Variante

Unter dem Gesichtspunkt der späteren Verwendung bei der Realisierung eines Ansatzes zur Endnutzerprogrammierung ist die zweite Variante die wichtigere, die erste Variante ist nur in wenigen, sehr speziellen Anwendungsszenarien von Bedeutung. Im Rahmen des prototypisch implementierten Übertragungsmechanismus wird das Verhalten der zweiten Variante realisiert. Sollte die erste Variante in einem Anwendungsszenario erforderlich sein, lässt sich das Verhalten durch eine entsprechende Komponente auch unter Verwendung des entworfenen Übertragungsmechanismus realisieren. Daher resultiert keine Einschränkung aus der an dieser Stelle getroffenen Entwurfsentscheidung.

7.3.3 Mehrere Sender, ein Empfänger

Neben den oben beschriebenen Unterscheidungen hinsichtlich der Anzahl der Empfänger ist es auch möglich, dass mehrere Sender Nachrichten an einen Empfänger (N:1, **many-to-one**) senden. Eine typische Anwendung für dieses Muster wäre beispielsweise die Realisierung einer gemeinsamen Protokollierung der Nachrichten für alle beteiligten Sender. Die unten stehende Abb. 7.5 verdeutlicht dieses Kommunkationsmuster.

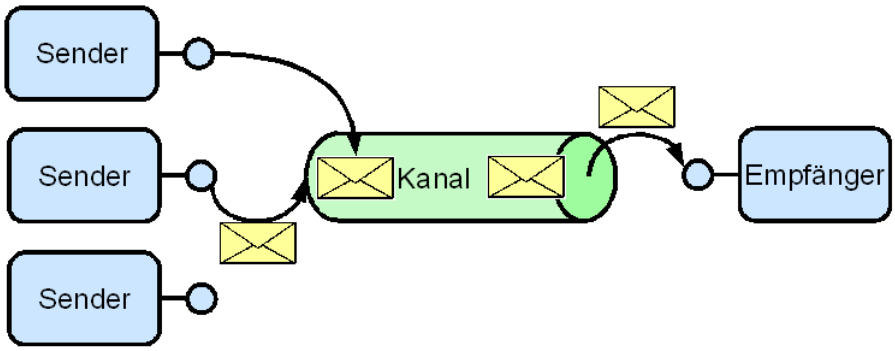


Abb. 7.5: Kommunikation zwischen mehreren Sendern und einem Empfänger

Auch dieses Szenario sollte sich auf Basis des zu entwerfenden Übertragungsmechanismus abbilden lassen.

Der Fall many-to-many muss nicht gesondert betrachtet werden, da er aus Sicht eines an der Kommunikation als Sender oder Empfänger Beteiligten immer auf die Fälle one-to-many beziehungsweise many-to-one zurückgeführt werden kann.

7.3.4 Weitere Differenzierungen

Neben den oben diskutierten Differenzierungen nach der Anzahl der Sender respektive Empfänger lassen sich weitere Eigenschaften einer nachrichtenbasierten Komposition betrachten (vgl. auch [Cha04]). So ist es auch möglich, Kommunikationsszenarien nach dem Verwendungszweck der Nachricht zu unterscheiden, hier existieren im Wesentlichen vier Varianten.

1. Die Nachricht stellt einen Aufruf dar, ihr Ziel ist es, eine Methode einer anderen Komponente aufzurufen. Dieser Nachrichtentyp entspricht einem entfernten Methodenaufruf (Remote Procedure Call, RPC).
2. Die Nachricht dient der Übermittlung von Daten zwischen Sender und Empfänger.
3. Die Nachricht informiert die Empfänger über das Auftreten eines Ereignisses.
4. Es handelt sich um ein Anfrage-Antwort-Szenario. Dies entspricht in der Realisierung dem Austausch je einer „normalen“ Nachricht in Hin- beziehungsweise Rückrichtung. Konzeptionell handelt es sich um eine andere Form der Kommunikation als in dem bisher besprochenen Fall einer Nachricht, auf die keine Antwort beziehungsweise Bestätigung erfolgt.

Im Zusammenhang mit der technischen Realisierung des Programmiermodells sind diese Unterscheidungen wichtig, da die miteinander kommunizierenden Komponenten auch in dieser Hinsicht kompatibel sein müssen. Bei der Realisierung des Nachrichtenkanals darf der Verwendungszweck der Nachricht ebenfalls nicht außer Acht gelassen werden, da sich hieraus unter Umständen besondere Anforderungen ergeben. So muss ein für den Transport von Datennachrichten vorgesehener Kanal beispielsweise Nachrichten entsprechender Größe zulassen, diese dürfte im Allgemeinen über der für Ereignisbenachrichtigungen benötigten Größe liegen.

Zusätzlich zu den Unterscheidungen nach dem Zweck der Nachrichten kann prinzipiell auch nach der Gestaltung des Übertragungskanals unterschieden werden. So sind spezialisierte Kanäle für einzelne Nachrichtentypen denkbar oder im Falle von Datennachrichten (Fall 2) für unterschiedliche Datentypen. Diese Unterscheidungen sind aber eher für die Umsetzung und weniger für den Entwurf des Programmiermodells von Belang.

7.4 Entwurf eines Kommunikationsmechanismus

Nach diesen grundlegenden Überlegungen zur Kommunikation zwischen Komponenten wird die Realisierung der Nachrichtenkanäle durch Warteschlangen (siehe Abschnitt 7.4.1) angesprochen. Im Anschluss daran wird ein Ansatz vorgestellt, welcher eine Kommunikation zwischen einer Menge von Komponenten ermöglicht und insbesondere die Umsetzung der vorgestellten Kommunikationsszenarien erlaubt. Dieser verwendet die zuvor vorgestellten Nachrichtenwarteschlangen.

Die Übermittlung von Nachrichten kann – wie oben bereits angesprochen – grundsätzlich sowohl synchron als auch asynchron erfolgen. Wird der Nachrichtenkanal mit einem Zwischenspeicher, beispielsweise in Form einer FIFO-Warteschlange, versehen, wird eine zeitliche Entkopplung von Sender und Empfänger möglich. Dadurch können auch solche Anwendungen realisiert werden, in denen pro Zeiteinheit mehr Nachrichten bei einem Empfänger eintreffen, als von diesem verarbeitet werden können. Der Zwischenspeicher muss jedoch nicht notwendigerweise in den Übertragungskanal integriert werden. Das resultierende Verhalten ließe sich ebenso durch die Einführung spezieller Pufferkomponenten in das Modell erreichen.

Im Rahmen der vorliegenden Arbeit wird eine asynchrone und gepufferte Kommunikation zwischen den einzelnen Komponenten verwendet, da diese Variante eine sehr lose Kopplung zwischen den Kommunikationspartnern ermöglicht.

7.4.1 Warteschlangenbasierte Nachrichtenkanäle

Eine nachrichtenbasierte Architektur führt nicht notwendigerweise zum Einsatz von Warteschlangen als zusätzliche Schicht zwischen den Kommunikationspartnern. Bei einem Großteil der heute eingesetzten nachrichtenbasierten Systeme erfolgt die Kommunikation zwischen den Beteiligten direkt (vgl. auch [CDK02], [IBM03]). Daher wird in diesem Abschnitt kurz die im Rahmen der vorliegenden Arbeit verwendete Kommunikation auf Basis von Nachrichtenwarteschlangen dargestellt.

Das durch den Einsatz von Nachrichtenwarteschlangen erreichte Verhalten lässt sich mit dem durch die E-Mail-Kommunikation zwischen Personen bekannten Verhalten vergleichen. Um eine E-Mail zu versenden oder zu empfangen, ist es nicht notwendig, dass der jeweils andere Kommunikationspartner in diesem Moment erreichbar ist. Etwas vereinfacht betrachtet schickt der Sender eine Nachricht nicht an den Empfänger, sondern an einen Zwischenspeicher. Dort wird eine gesendete Nachricht so lange gespeichert, bis der Empfänger sie abholt. Da der Zwischenspeicher in der Regel unabhängig vom Sender ist, kann das Abholen geschehen, ohne dass der Sender in diesem Moment verfügbar sein muss. Durch den Einsatz von Nachrichtenwarteschlangen kann dieses Verhalten auf die Kommunikation von Applikationen übertragen werden.

Eine Nachrichtenwarteschlange basiert auf dem abstrakten Datentyp Queue. Die gängigen Realisierungen verfügen neben dieser Grundfunktionalität über Erweiterungen wie beispielsweise Möglichkeiten zum Priorisieren von Nachrichten oder Transaktionskonzepte für die Übermittlung von Nachrichten (siehe zum Beispiel [IBM06] [Mic06j]).

Je nach Implementation hält die Nachrichtenqueue die Nachrichten auch dann vor, wenn der Empfänger noch nicht oder nicht mehr mit der Queue verbunden ist. Dadurch wird es möglich eine Nachricht zu versenden, ohne dass es nötig ist, darauf zu achten, dass der Empfänger in diesem Moment in einem empfangsbereiten Zustand ist. Es handelt sich also um eine asynchrone Form der Kommunikation. Durch die Zwischenspeicherung können Verzögerungen in der Kommunikation – beispielsweise für den Fall, dass in einer Zeiteinheit mehr Nachrichten eintreffen, als verarbeitet werden können – abgefangen werden. Dieser Puffer verhindert, dass die Gesamtanwendung in einem

solchen Fall in Mitleidenschaft gezogen wird. Die Verwendung der Nachrichtenwarteschlangen führt insgesamt zu einer verlässlicheren Kommunikation, da ein Ausfall eines an der Kommunikation Beteiligten nicht notwendigerweise Auswirkungen auf das Gesamtsystem haben muss. Zu beachten ist jedoch, dass die Verwendung der Nachrichtenwarteschlangen sich negativ auf die Performanz auswirken kann und eine zusätzliche Fehlerquelle in die Anwendung einführt.

Eine Nachrichtenwarteschlange ist für den Transport von Nachrichten und gegebenenfalls für deren kurzfristige Zwischenspeicherung, nicht jedoch für deren Langzeitspeicherung geeignet. Für solche Anwendungen müssen zusätzliche Mechanismen realisiert werden, zum Beispiel auf Basis eines Datenbanksystems. Warteschlangen verfügen insbesondere nicht über ein Abfragesystem für Nachrichten. Eine Speicherung von Nachrichten in der Warteschlange birgt also das Risiko, einen Datenpool zu schaffen, der nicht in geeigneter Art und Weise zugreifbar ist.

Dadurch, dass die Kommunikation über Nachrichtenwarteschlangen asynchron ist, kann die sendende Komponenteninstanz direkt nach dem Senden bereits mit ihrer Verarbeitung fortfahren, unabhängig davon, ob der Empfänger die Verarbeitung der Nachricht bereits abgeschlossen hat oder nicht.

Eigenschaften der Warteschlangen, die über die eben dargestellten (asynchron und gepuffert) hinausgehen – wie beispielsweise Ansätze zur Priorisierung von Nachrichten oder zur Zusammenfassung von einzelnen Nachrichten zu einer Transaktion – finden in der prototypischen Realisierung keine Verwendung. Aus diesem Grund werden diese weitergehenden Möglichkeiten des Warteschlangenkonzeptes hier nicht vertieft. Die verwendeten Eigenschaften der Nachrichtenwarteschlangen sind nicht abhängig von einer bestimmten Implementation bzw. einem speziellen Produkt.

7.4.2 Nachrichtenvermittler

Der in diesem Abschnitt vorgestellte Ansatz beschreibt auf einer hohen Abstraktionsebene den Entwurf einer Anwendung, mit deren Hilfe sich eine nachrichtenbasierte Kommunikation zwischen einer Menge von Komponenten realisieren lässt. Diese Applikation wird als Cobamos Nachrichtenvermittler bezeichnet und findet Verwendung zur Komposition von Anwendungen aus einzelnen Komponenten.

Prinzipiell könnte die Kommunikation zwischen zwei Komponenteninstanzen auch auf Basis einer einzelnen Nachrichtenqueue realisiert werden. Eine der beiden Instanzen sendet an die Warteschlange, die zweite Instanz empfängt ihre Nachrichten aus dieser Warteschlange. Dies ist im Falle der direkten Kommunikation zwischen zwei Instanzen ein denkbarer Weg. Aber schon im Falle einer Kommunikationsstruktur, bei der ein Sender Nachrichten an viele parallele Empfänger senden soll, reicht dieser naive Ansatz nicht mehr aus.

Die zunächst vielleicht nahe liegende Idee, eine der Anzahl der Empfänger entsprechende Anzahl von Kopien der Nachricht in die Warteschlange zu senden, scheitert aus zwei Gründen. Zum einen ist dem Sender nicht bekannt, wie viele Konsumenten sich auf der anderen Seite der Warteschlange befinden, zum anderen ist nicht sichergestellt, dass die Implementation der Queue dafür Sorge trägt, dass die Konsumenten reihum auf die Schlange zugreifen. Die Zugriffsreihenfolge ist insbesondere beim asynchronen Empfang nichtdeterministisch und es kann nicht sichergestellt werden, dass jeder Empfänger eine Nachricht erhält. Eine direkte Adressierung der Empfänger würde die Kopplung zwischen den an der Kommunikation beteiligten Komponenten erhöhen. Der Ansatz, solange zu senden bis jeder Empfänger den erfolgreichen Empfang signalisiert, führt einerseits zu einem enormen Overhead in der Kommunikation und Synchronisation und andererseits zu einer Vielzahl redundanter

Nachrichten, die auf Seiten der Empfänger auch als solche erkannt werden müssen. Hier sind also andere Lösungsansätze nötig.

Die Unterstützung für dieses und die restlichen in diesem Kapitel angesprochenen Kommunikationsmuster gehört zu den Anforderungen an den Cobamos Nachrichtenvermittler. Darüber hinaus sollte diese Anwendung so weit als möglich unabhängig von den für die Nachrichten verwendeten Sprachen sein, da der später vorgestellte Entwurf des Programmiermodells die Verwendung unterschiedlicher Sprachen unterstützt.

Die in Abb. 7.6 dargestellte Übersichtsgrafik verdeutlicht das Prinzip des realisierten Nachrichtenvermittlers. Die Applikation überwacht eine Reihe von Nachrichtenwarteschlangen auf neue Nachrichten. Dies sind die auf der linken Seite des Vermittlers dargestellten Warteschlangen. Dabei handelt es sich in der Regel um Ausgangswarteschlangen aus Sicht einer sendenden Komponenteninstanz (im Beispiel ist „Ausgang A“ die Ausgangswarteschlange der Komponenteninstanz A). Ein Port einer Komponenteninstanz ist in diesem Entwurf mit einer Warteschlange assoziiert. Je nach Richtung des Nachrichtenflusses für diesen Port sendet die Komponenteninstanz Nachrichten in die Warteschlange oder holt diese dort ab. Auf der rechten Seite des Vermittlers sind Nachrichtenqueues zu sehen, die aus Sicht der Komponenteninstanzen Eingänge repräsentieren.

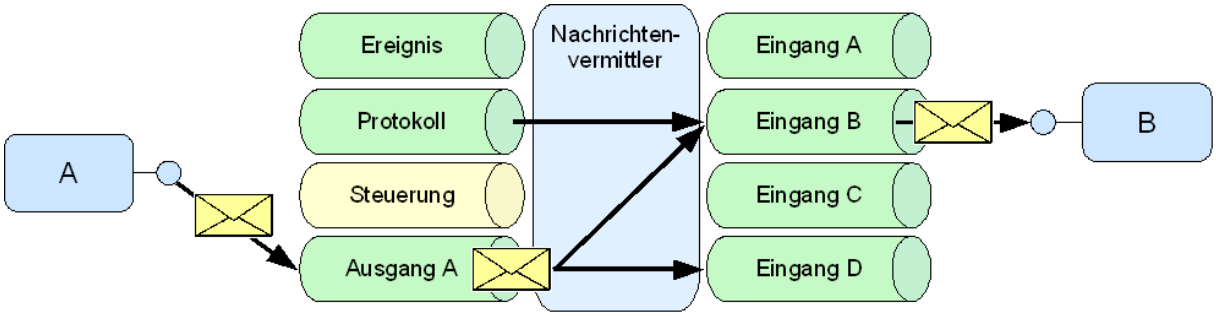


Abb. 7.6: Anwendungsidee des Nachrichtenvermittlers

Empfängt der Vermittler eine Nachricht in einer der überwachten Warteschlangen, so holt er diese ab und sendet sie an alle Nachrichtenwarteschlangen (die Eingangswarteschlangen der Komponenten), die als Interessenten für diese Eingangswarteschlange registriert sind (auf die genaue Realisierung dieser Registrierung wird weiter unten eingegangen). Das Verteilen der eintreffenden Nachrichten an Interessenten erfolgt ausschließlich auf Basis dieser Registrierungen, darüber hinausgehende Ansätze – wie beispielsweise ein Verteilen abhängig vom Inhalt der Nachrichten – sind in der prototypischen Realisierung nicht umgesetzt. Im Rahmen der Anwendungen erwies sich dieser Ansatz als ausreichend.

Einer Ausgangswarteschlange einer Komponente – also einer Eingangswarteschlange aus Sicht des Vermittlers – können mehrere Interessentenwarteschlangen zugeordnet werden. Des Weiteren kann ein Interessent auch für mehrere Nachrichtenquellen registriert sein, dadurch können die in 7.3 angesprochenen Kommunikationsszenarien umgesetzt werden.

Das Empfangen geschieht asynchron und ereignisgesteuert. Dies hat den Vorteil, dass die Verarbeitung des Nachrichtenvermittlers nicht blockiert während er auf das Eintreffen einer Nachricht in einer bestimmten Warteschlange wartet, sondern ständig Nachrichten verarbeiten kann. Da der Nachrichtenvermittler eine erhaltene Nachricht nicht direkt an eine Komponenteninstanz sondern an als Interessenten registrierte Ausgangswarteschlangen verteilt, kann der jeweilige Empfänger sie nach Bedarf aus der Warteschlange abholen und muss im Augenblick des Sendens nicht empfangsbereit

sein. Dadurch werden auch Anwendungen möglich, in denen nicht alle Empfänger einer Nachricht diese gleichzeitig empfangen müssen. Vielmehr kann der Teil der gerade empfangsbereiten Empfänger die aktuellen Nachrichten verarbeiten, ein später „hinzukommender“ Empfänger – z. B. nach einem Neustart, oder dem Austausch der Komponenteninstanz durch eine aktuellere Version – findet in seiner Warteschlange alle bis dahin an ihn adressierten Nachrichten und kann die Verarbeitung fortsetzen. Hier muss jedoch sorgfältig geprüft werden, ob dieses Verhalten gewünscht ist. Unter Umständen können beim Empfang von „älteren“ Nachrichten Inkonsistenzen auftreten. In einem solchen Fall müsste die Eingangswarteschlange einer Komponenteninstanz bei der Assoziierung geleert werden.

Eine Komponente muss in diesem Szenario nur „ihre“ Ein- und Ausgangswarteschlangen kennen. Weder der Nachrichtenvermittler noch die an der Kommunikation beteiligten Sender oder Empfänger müssen einer Komponente bekannt sein. Dieser Entwurf ermöglicht also die mehrfach angesprochene lose Kopplung zwischen den interagierenden Komponenteninstanzen. Die lose Kopplung einer Komponente mit ihrer Umgebung war auch eine der Forderungen der konzeptuellen Komponentendefinition aus Kapitel 3.

Da auf Basis der über den Nachrichtenvermittler kommunizierenden Komponenteninstanzen eine Gesamtanwendung realisiert werden soll, muss es eine Möglichkeit geben, komponentenübergreifende Verhaltensweisen zu realisieren. Aus diesem Grund existieren neben den Warteschlangen, die einer einzelnen Komponente oder genauer einem Komponentenein- bzw. -ausgang zugeordnet sind, auch gemeinsam genutzte Warteschlangen. Dies sind in der obigen Grafik die Warteschlangen „Ereignis“, „Protokoll“ und „Steuerung“, wobei letztere eine Sonderrolle einnimmt. Die Warteschlange „Ereignis“ dient zur Realisierung eines anwendungsweiten Mechanismus für Ereignisbenachrichtigungen. Da die Gesamtanwendung aus einer Reihe von Einzelkomponenten besteht, und diese soweit als möglich unabhängig voneinander sein sollen, gibt es zunächst keine Möglichkeit, dass ein Ereignis anwendungsweit kommuniziert werden kann. Um dies zu ermöglichen, wird jede Komponenteninstanz, die eine Eingangswarteschlange (aus Komponentensicht) beim Nachrichtenvermittler registriert, auch als Interessent für die Ereigniswarteschlange eingetragen. Eine Nachricht, die an diese Warteschlange gesendet wird, wird folglich an alle Komponenteninstanzen in der Anwendung übertragen. Dies ermöglicht beispielsweise ein gleichzeitiges Beenden aller Komponenteninstanzen, auch wenn diese in eigenen Prozessen ausgeführt werden und nicht direkt kommunizieren können. Auch die Benachrichtigung über Fehlerzustände, die die gesamte Anwendung betreffen, können auf diese Weise verteilt werden. Gegebenenfalls kann für diese Aufgabe auch eine eigene Warteschlange reserviert werden. Generell ist beim Einsatz von anwendungsweiten Warteschlangen in der Art der eben beschriebenen Ereigniswarteschlange zu beachten, dass ein Sender eine von ihm versandte Nachricht ebenfalls empfängt. Dies kann unter Umständen zu inkonsistentem Verhalten führen und muss deswegen bei der Realisierung berücksichtigt werden. Abhilfe kann hier schon dadurch geschaffen werden, dass eine Nachricht zur Ereignisbenachrichtigung den ursprünglichen Sender als Absender enthält. Beim Empfang einer solchen Nachricht kann eine Komponenteninstanz dann prüfen, ob sie die ursprüngliche Quelle dieser Benachrichtigung ist und dementsprechend reagieren.

Die Warteschlange „Protokoll“ realisiert ebenfalls eine solche anwendungsübergreifende Querschnittsaufgabe. Wenn innerhalb der Anwendung Fehler oder Ähnliches protokolliert werden sollen, ist es unzweckmäßig, dass jede Komponenteninstanz ein eigenes Protokoll führt. Diese müssten alle getrennt ausgewertet werden oder es müssten geeignete Maßnahmen zur Zusammenführung der Protokolldaten umgesetzt werden. Da der Nachrichtenvermittler in jeder Anwendung vorhanden ist, liegt es nahe, die Verantwortung für die Protokollierung hier anzusiedeln. Und um die lose Kopplung zwischen Komponenten und Vermittler zu erhalten, erfolgt die Interaktion mit dem Protokollierungsmechanismus ebenfalls über Nachrichtenwarteschlangen.

Dieser letzte Aspekt liefert auch die Begründung für die Existenz der Warteschlange „Steuerung“. Bei der Erläuterung der grundlegenden Funktionsweise des Nachrichtenvermittlers wurde bereits die Registrierung von Interessenten für Warteschlangen angesprochen. Zum einen sollte eine Komponenteninstanz die Möglichkeit haben, sich selbst als Interessent für eine Nachrichtenquelle zu registrieren. Zum anderen gibt es auch Einsatzfelder, in denen die Verknüpfung von eingehenden und ausgehenden Warteschlangen unabhängig von den sendenden und empfangenden Komponenteninstanzen – beispielsweise durch eine andere Applikation wie die Entwicklungsumgebung – vorgenommen werden muss. Zum Erzeugen der Verknüpfungen muss Funktionalität aufgerufen werden, welche in der Verantwortlichkeit des Nachrichtenvermittlers liegt. Bei diesen Zugriffen sollte nach Möglichkeit die bisher in diesem Entwurf erreichte lose Kopplung zwischen den Bestandteilen des Anwendungssystems beibehalten werden. Daher liegt es nahe, die benötigte Funktionalität über Nachrichten aufzurufen. Die Möglichkeiten hierzu sind als kleinster Nenner aller Komponenten, welche unter Verwendung des Nachrichtenvermittlers kommunizieren sollen, vorhanden. Eine außerhalb der Komponentenkomposition stehende Applikation, welche die Verknüpfungen konfigurieren soll, kann ebenfalls über Nachrichten mit dem Vermittler kommunizieren. Auch für diese Interaktion wird dann eine weitgehende Unabhängigkeit der beteiligten Entitäten erreicht, da keine direkten Referenzen notwendig sind.

In Abschnitt 8.7.1 wird detaillierter auf die vorgenommene prototypische Realisierung des Nachrichtenvermittlers eingegangen. Den Abschluss dieses Kapitels bildet eine kurze Behandlung der Vor- und Nachteile, die aus der Verwendung einer nachrichtenbasierten Kommunikation im hier betrachteten Zusammenhang resultieren.

7.5 Vorteile einer nachrichtenbasierten Komposition

Wenn man die Vorteile einer nachrichtenbasierten Komposition von Komponenten betrachtet, ist zunächst die lose Kopplung der Komponenten zu erwähnen, die auf diese Weise erreicht werden kann. Eine Komponente ist in diesem Ansatz vom verwendeten Nachrichtenkanal abhängig, nicht jedoch von einer konkreten Komponente als Empfänger oder Sender. Instanzen jedes Komponententyps, die in der Lage sind, Nachrichten im korrekten Format zu erstellen beziehungsweise zu verarbeiten, können in diesen Rollen auftreten. Je nach Realisierung des Nachrichtenkanals ist noch nicht einmal eine gemeinsame Programmiersprache oder Plattform nötig. Auch muss zum Kommunikationszeitpunkt keine Referenz auf den Kommunikationspartner existieren.

Durch diese lose Kopplung ist eine über die späte Bindung, wie sie in der Objektorientierung existiert, hinausgehende Flexibilität möglich. Bei der späten Bindung der Objektorientierung steht der Empfänger einer Nachricht beziehungsweise eines Methodenaufrufs nicht zur Übersetzungszeit, sondern erst zum Zeitpunkt des Aufrufes beziehungsweise des Sendens während der Laufzeit einer Anwendung fest. In einer nachrichtenbasierten Architektur kann das Senden einer Nachricht erfolgen, ohne dass der konkrete Empfänger feststeht. Dies ist besonders elegant unter Verwendung von Nachrichtenwarteschlangen (siehe Abschnitt 7.4.1) möglich. Diese Möglichkeit kann als sehr späte Bindung (very late binding) bezeichnet werden.

Durch die Eigenständigkeit der Komponenten und deren lose Kopplung kann ein Softwaresystem derart gestaltet werden, dass ein Ausfall einzelner Komponenten nicht notwendigerweise zu einem Versagen des Gesamtsystems führt. Allerdings muss hier im konkreten Anwendungsfall sorgfältig geprüft werden, ob der verbleibende Teil der Anwendung sinnvoll nutzbar ist und die Ergebnisse noch ausreichend vertrauenswürdig sind. Anderenfalls ist ein geordnetes Beenden der Applikation zu empfehlen.

Weitere Vorteile der Nachrichtenbasierung ergeben sich im Entwicklungsprozess derartiger Systeme. Die lose Kopplung der beteiligten Komponenten erleichtert in erheblichem Maße die getrennte Entwicklung. Für die Phase des Testens ergeben sich ebenfalls Vorteile. So sind Komponenten gemäß ihrer Definition abgeschlossen gegenüber ihrer Umgebung und eignen sich daher als Kapselung für das so genannte Unit-Testen (siehe beispielsweise [SL05]). Im Falle von Komponenten für den Einsatz in einem nachrichtenbasierten Softwaresystem können diese Tests auf das Generieren geeigneter synthetischer Nachrichten zur Übermittlung an die zu testende Komponente und das Aufzeichnen und Überprüfen der von ihr gesendeten Nachrichten reduziert werden. Auch für die Evolution der Gesamtanwendung hat das hier diskutierte Paradigma Vorteile, da aufgrund der losen Kopplung einzelne Komponenten vergleichsweise einfach gegen neuere Versionen ausgetauscht werden können. Die Möglichkeit einzelne Nachrichten zu untersuchen erleichtert die Fehlersuche – in diesem Zusammenhang ist auch die im anschließenden Abschnitt 7.6 angesprochene Aufzeichnung von Nachrichten zur Laufzeit einer Anwendung von Interesse – während der Entwicklung einer Applikation. Allerdings ist die Lesbarkeit im Sinne des Verstehens durch einen menschlichen Konsumenten realisierungsabhängig. Eventuell sind hierfür unterstützende Anwendungen zur Visualisierung der Nachrichtendaten nötig.

7.6 Ausblick auf weitere Möglichkeiten

In diesem Abschnitt werden einige denkbare Erweiterungen angesprochen, welche sich auf Basis des nachrichtenorientierten Ansatzes realisieren lassen. Diese Ideen sind zum Großteil noch nicht in die prototypische Realisierung eingeflossen, auch erhebt diese Aufzählung keinen Anspruch auf Vollständigkeit. Vielmehr sollen einige interessante Möglichkeiten aufgezeigt werden, welche unter Umständen auch Impulse für weitere Arbeiten in diesem Bereich liefern.

Durch die Betonung der Nachrichten als eigenständige Entitäten in der Realisierung der Anwendung ergeben sich weitere Möglichkeiten. So können beispielsweise Filter in den Nachrichtenkanälen eingesetzt werden, welche nur bestimmte Nachrichten passieren lassen oder den Nachrichteninhalt manipulieren oder passierende Nachrichten noch auf einen anderen Übertragungskanal umleiten. Filter dieser Art erlauben es, Vorverarbeitungen in einer Anwendung einzuführen, ohne dass die Komponenten zu diesem Zweck angepasst oder explizite Umleitungen des Nachrichtenflusses vorgenommen werden müssten. Eine solche Funktionalität wurde im Rahmen der prototypischen Umsetzung dieses Konzeptes integriert, Näheres hierzu findet sich in Abschnitt 8.7.1.

Eng mit dem Einsatz von Filtern verwandt ist die Realisierung eines nachrichtenabhängigen Steuerflusses innerhalb der Anwendung. Hierunter ist zu verstehen, dass der Weg, den eine Nachricht durch die Anwendung nimmt, abhängig von den Daten in der Nachricht ist. Dabei kommen zum einen Entscheidungen auf Basis von Daten in Frage, die eine Bedeutung in der Anwendung haben. Ein Beispiel wäre die Prüfung auf das Einhalten eines Grenzwertes und ein entsprechender Eskalationsweg für den Fall einer Überschreitung. Zum anderen können in den Nachrichten spezielle Datenfelder eingeführt werden, welche nur zur Steuerung des Kontrollflusses dienen. Mit diesen kann dann beispielsweise eine Lebensdauer für die Nachricht festgelegt werden. Dies kann – ähnlich der beim Internet Control Message Protocol (ICMP) verwendeten Zähler für die Anzahl der passierten Rechner, siehe [Mal93] – geschehen, indem man die Anzahl der Komponenten festlegt, die eine Nachricht „durchqueren“ darf. Alternativ kann auch ein simples Zeitstempelverfahren eingesetzt werden. Eine weitere Möglichkeit, die die Verwendung spezieller Metainformationen in den Nachrichten eröffnet, ist die Bereitstellung von Versionsinformationen. Dies ermöglicht beispielsweise in der Phase der Evolution eines Softwaresystems den parallelen Betrieb von alten und neuen

Komponenten und eine Zuteilung der Nachrichten anhand ihrer Versionsinformationen. Solche nachrichtenabhängigen Verzweigungen des Steuerflusses können sowohl durch spezielle Komponenten als auch durch Nachrichtenkanäle realisiert werden. Die gewählte Form der Realisierung führt zu einer anderen Betrachtung beim Modellieren, da die Funktionalität, wenn sie in einer eigenen Komponente bereitgestellt wird, stärker betont wird.

Eine weitere interessante Möglichkeit ist die – am Beispiel der Sprache Ruby bereits vorgestellte – Aufzeichnung von Nachrichten. Je nach Realisierung der Nachrichtenkanäle – in der prototypisch realisierten Applikation ist eine entsprechende Schnittstelle vorhanden – kann die Speicherung der übermittelten Nachrichten in einer Datenbank erfolgen. Dies ermöglicht beispielsweise Auswertungen über eine Anwendung beziehungsweise die Kommunikation zwischen ihren Komponenten. Handelt es sich um eine Anwendung aus dem Bereich des E-Learning und stellen die Komponenten einzelne Lehreinheiten respektive Übungsmöglichkeiten dar, so können aufgrund der Nachrichten unter anderem Verweildauern oder besonders häufig gewählte Alternativen bestimmt werden. Diese Informationen erlauben Rückschlüsse auf das Lernverhalten und ermöglichen die Realisierung adaptiver Lernanwendungen. Als weitere Möglichkeit ergibt sich die Restauration des Systemzustandes zu einem beliebigen Zeitpunkt. Dies gilt unter der Voraussetzung, dass der Zustand eines Systems zu einem bestimmten Zeitpunkt durch die zuvor zwischen den Komponenten ausgetauschten Nachrichten bestimmt ist. Abhängig von der Granularität der Nachrichten erschließen sich auf Basis dieser Aufzeichnungen interessante Anwendungen. In [Sch03] wird beispielsweise ein System zur Aufzeichnung und Wiedergabe von Nutzerinteraktionen zur Verwendung im Bereich des E-Learning dargestellt. Dieses System arbeitet jedoch auf Basis der Interaktionsnachrichten der Benutzungsoberfläche, das heißt, es werden alle Mausbewegungen etc. aufgezeichnet. Dieser niedrige Abstraktionslevel führt zu einem sehr hohen Datenaufkommen. Die Granularität der Komponentenübergänge und damit die Menge der zu beobachtenden Nachrichten sollte für eine Anwendung in diesem Bereich höher angesiedelt sein.

Wird eine Anwendung zur Simulation eines dynamischen Modells auf Basis von Komponenten und Nachrichten aufgebaut, so ist es möglich, das System schrittweise zu betrachten und Berechnungen zu einem bestimmten Zeitpunkt „anzuhalten“. Dies erleichtert insbesondere bei dynamischen Systemen die Analyse und das Verständnis ganz erheblich.

Der nachrichtenbasierte Ansatz überträgt sich in natürlicher Weise auf verteilte Systeme. Durch die Entkopplung, welche der Nachrichtenkanal bewirkt, ist es für Sender respektive Empfänger unerheblich, ob der jeweils andere Kommunikationspartner auf derselben oder einer entfernten Maschine ausgeführt wird. Eng mit dieser Möglichkeit verknüpft ist das Potenzial dieses Ansatzes hinsichtlich der Überwindung von Plattformgrenzen. Hierunter fallen Programmiersprachen- und Betriebssystemgrenzen. Werden die Nachrichten beispielsweise in einem XML-basierten Format definiert und stehen Nachrichtenkanäle bereit, welche – zum Beispiel auf Basis von verbreiteten Technologien wie dem Hypertext Transfer Protocol (HTTP) – diese Nachrichten über Plattformgrenzen hinweg transportieren können, so sind diese im Falle eines herkömmlichen Anwendungsentwurfs einschränkenden Grenzen kein Hindernis. Ansätze zur Überwindung von Plattformgrenzen, welche auf Aufrufkonventionen basieren um eine gewisse Kompatibilität des ausführbaren Codes zu erreichen, sind im Allgemeinen komplizierter umzusetzen und weniger robust[69].

Ein interessanter Aspekt ist die Verarbeitung von Nachrichten, welche nur zum Teil interpretiert werden können. Diese Situation kann entstehen, wenn syntaktische Fehler in einer Eingangsnachricht existieren. Dieser Fall tritt aber auch dann auf, wenn das Vokabular der Eingangsnachricht umfang-

[69] Ein solcher Ansatz findet sich beispielsweise in Microsofts COM Technologie, siehe [GT00].

reicher ist als das von der empfangenden Komponente interpretierbare. Um ein zuverlässiges und sicheres Softwaresystem zu realisieren, darf die Verarbeitung durch den Empfänger in dieser Situation nicht durchgeführt werden und es muss eine geeignete Fehlerbehandlung stattfinden. Falls der für die Anwendung notwendige Grad an Zuverlässigkeit es erlaubt, könnte der Empfänger alternativ dazu die Teile der Nachricht verarbeiten, die für ihn verständlich sind. Eine entsprechende Markierung an den produzierten Nachrichten könnte deren Unzuverlässigkeit anzeigen respektive unzuverlässige Teilbereiche markieren. Hier wäre in weiteren Untersuchungen zu klären, ob es möglich ist Gütegrade für die korrekte Erkennung von Nachrichten zu definieren und dementsprechend auch Grenzwerte für die Zuverlässigkeit von Ergebnissen festzulegen.

Einige der in diesem Abschnitt angesprochenen Möglichkeiten lassen sich auch auf Basis anderer Ansätze realisieren, dies gilt in ähnlicher Weise für die Vorteile. Der nachrichtenorientierte Ansatz ermöglicht jedoch meines Erachtens einen sehr natürlichen Zugang und eine elegante Realisierung.

7.7 Nachteile und Probleme der nachrichtenbasierten Komposition

Auch im Falle der nachrichtenbasierten Komposition bestehen einige Nachteile beziehungsweise Einschränkungen. Diese sollen im Folgenden untersucht werden.

Im Falle einer asynchronen Kommunikation können aufgrund der unbestimmten zeitlichen Abfolge verschiedene Komplikationen auftreten (siehe auch [SGM03]). Betrachtet man beispielsweise die in Abb. 7.7 dargestellte Situation, in welcher der Sender *A* eine Nachricht *e1* an *B* und *C* sendet. *B* sendet sofort bei Empfang der Nachricht *e1* eine Nachricht *e2* an *D* und *C*. Dann ist die Reihenfolge, in der die Nachrichten *e1* und *e2* von *C* empfangen werden, nicht definiert.

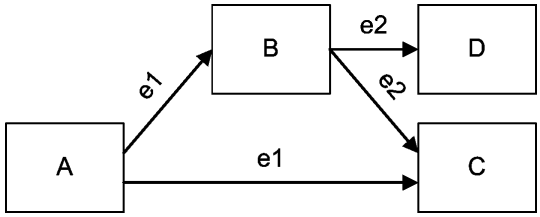


Abb. 7.7: Kommunikationsstruktur zwischen den Entitäten A, B, C und D

Es ergeben sich zwei unterschiedliche Reihenfolgen für das Eintreffen der Nachrichten bei *C*. Diese sind in Abb. 7.8 gegenübergestellt. Die obere Reihenfolge wird als **natürliche Reihenfolge** bezeichnet. Sie entspricht – verdeutlicht man sich den Nachrichtenfluss in Form eines Baumes – einer Breitensuche. Die zweite Reihenfolge entspricht hingegen einer Tiefensuche innerhalb des Baumes. Da diese Reihenfolge auch dem rekursiven Aufruf entspricht, wird sie als **rekursive Reihenfolge** bezeichnet.

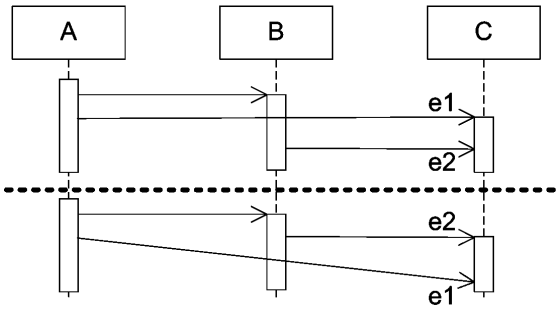


Abb. 7.8: Natürliche und rekursive Reihenfolge des Nachrichtenempfangs

Wichtig ist die Betrachtung dieser Szenarien, da sie häufig der Grund für schwer zu findende Fehler beziehungsweise unerwartetes Systemverhalten sind. Betrachtet man das Kommunikationsszenario als Ereignisbenachrichtigung und *e2* als ein durch *e1* ausgelöstes Ereignis, so wird klar, dass *C* im Falle der rekursiven Reihenfolge eventuell in einem inkonsistenten Zustand ist und nur die natürliche Reihenfolge die Kausalität der Nachrichtenreihenfolge erhält. Falls das Szenario das Einleiten einer Transaktion beschreibt und die entsprechenden Ressourcen gesperrt werden, so ist unter Umständen die rekursive Reihenfolge die gewünschte (ebenso wenn *B* beispielsweise mit *e2* eine Autorisierung an *C* sendet, damit *A e1* ausführen darf). Falls im Rahmen einer Anwendung die Erhaltung der natürlichen Reihenfolge oder zwingend die rekursive Reihenfolge notwendig ist, so müssen geeignete Maßnahmen implementiert werden, um dies sicherzustellen. Eine genauere Betrachtung findet sich in [CDK02]. Im Rahmen der prototypischen Realisierungen ergaben sich diese Notwendigkeiten nicht, so dass hier keine weitergehenden Untersuchungen vorgenommen wurden.

Generell ist im Falle nachrichtenbasierter Kommunikation die Situation denkbar, dass der Empfänger einer Nachricht sich bei ihrem Eintreffen in einem inkonsistenten Zustand befindet. Beim Entwurf der beteiligten Kommunikationspartner und der Nachrichtenarchitektur muss also dafür Sorge getragen werden, dass solche Situationen entweder vermieden werden können oder geeignete Mechanismen zu deren Behandlung bereitstehen. Verbreitete Ansätze hierfür basieren auf der Verwendung von Verzögerungen. Dabei wird eine Nachricht zwischengespeichert und erst übermittelt, nachdem der Empfänger einen konsistenten Zustand erreicht hat. Bei der Verwendung von Nachrichten-warteschlangen ist es beispielsweise möglich, dass ein Empfänger für die Zeitspanne, in der er sich in einem inkonsistenten oder generell nicht verarbeitungsbereiten Zustand befindet, das Empfangen der Nachrichten aussetzt. Diese werden dann bis zur erneuten Verarbeitungsbereitschaft des Empfängers in der Queue zwischengespeichert. Ein solches Aussetzen der Verarbeitung wurde auch bei der Realisierung der prototypischen Komponenten für die Beispielanwendungen (siehe Kapitel 9) implementiert.

Ein weiterer Punkt, der besondere Beachtung verdient, ist der Umgang mit Nachrichten, die vom Empfänger nicht interpretiert werden können. Ein direktes Übertragen des gängigen Verhaltens eines objektorientierten Systems – also ein Fehler, der die Beendigung der Gesamtanwendung zur Folge hat – ist nicht immer sinnvoll[70]. Handelt es sich beispielsweise um ein Multicast-Szenario, so ist es denkbar, dass die übrigen Empfänger der Nachricht diese ohne Fehler interpretieren konnten. Dies kann durch den vom Fehler betroffenen Empfänger jedoch nicht ohne weiteres festgestellt werden.

[70] Es sind natürlich auch Anwendungsszenarien (z. B. Steuerungssoftware für Anlagen) denkbar, in denen selbst bei einem lokalen Fehler ein geordnetes Beenden des Gesamtsystems erforderlich werden kann. Prinzipiell bietet der vorgestellte Entwurf des Nachrichtenvermittlers mit den „gemeinsamen“ Warteschlangen (z. B. Ereignis) auch für diesen Fall einen Ansatz, dieser ist aber für diese Verwendung nicht bis zum produktiven Stadium umgesetzt worden.

Darüber hinaus soll zum einen die lose Kopplung der Komponenten gewahrt bleiben, andererseits besteht die Notwendigkeit einer komponentenübergreifenden Fehlerbehandlung für die Gesamtanwendung. Dies ist auch vor dem Hintergrund wichtig, dass für den späteren Nutzer einer solchen Applikation der Eindruck eines geschlossenen Gesamtsystems entstehen soll. Im Rahmen des Entwurfs für den Nachrichtenvermittler bieten die anwendungsübergreifenden Nachrichtenwarteschlangen einen Lösungsansatz für diese Problematik. Die gemeinsame Fehlerbehandlung wird – wie die anderen anwendungsübergreifenden Aufgaben auch – beim Nachrichtenvermittler angesiedelt und über eine dedizierte Nachrichtenwarteschlange angesprochen.

Ein weiterer Nachteil der nachrichtenbasierten Komposition ist, dass sie kein natürliches Kapselungskonzept für Kompositionen mit sich bringt. Um ein Zusammenfassen mehrerer Komponenten zu einer einzigen oder zu einem anderen Kapselungskonzept einer höheren Abstraktionsebene zu realisieren, sind über die nachrichtenbasierte Komposition hinausgehende Ansätze nötig.

Die notwendige Einführung von Nachrichtenkanälen kann die Performanz des Gesamtsystems im Vergleich zu einer direkten Kommunikation beeinträchtigen. Die Anzahl der Übergänge zwischen den Komponenten sollte daher im Vergleich mit den Aufrufstrukturen in einem herkömmlichen Programm vergleichsweise gering sein. Komplexe und langwierige Berechnungsvorschriften sollten nach Möglichkeit innerhalb einer Komponente realisiert sein und nicht die Interaktion mehrerer Komponenten erfordern.

7.8 Fazit

In diesem Kapitel wurde ein Konzept zur Komposition von Komponenten auf Basis von Nachrichten motiviert. Die grundlegenden Möglichkeiten der Ausgestaltung dieses Ansatzes und die zu unterscheidenden Kommunikationsstrukturen wurden vorgestellt, dies geschah bereits im Hinblick auf die notwendigen Anforderungen für eine Realisierung eines entsprechenden Programmiermodells zur Nutzung im Cobamos-Framework.

Diese Realisierung wird im folgenden Kapitel vorgenommen. Dadurch wird zum einen die Umsetzbarkeit der in diesem Kapitel angestellten Überlegungen gezeigt. Zum anderen handelt es sich um ein umfangreicheres Beispiel für eine Umsetzung des MVC'-Musters und der übrigen Konzepte aus den vorangegangenen Kapiteln dieser Arbeit.

8 Realisierung eines nachrichtenbasierten Programmiermodells

„Der Wissenschaftler arbeitet, wie alle Organismen, mit der Methode von Versuch und Irrtum. Der Versuch ist eine Problemlösung. Der Irrtum, oder genauer die Irrtumskorrektur, ist in der Evolution des Pflanzen- und Tierreichs gewöhnlich die Ausmerzung des Organismus; in der Wissenschaft die Ausmerzung der Hypothese oder Theorie.“
Karl Raimund Popper, Philosoph und Wissenschaftstheoretiker, 1902 – 1994

In diesem Kapitel werden Aspekte der Realisierung des in Kapitel 7 vorgestellten Programmiermodells zur nachrichtenbasierten Komposition von Komponenten vorgestellt. Diese Umsetzung erfolgt unter Verwendung des Cobamos-Frameworks und stellt damit eine Anwendung der Überlegungen der vorangegangenen Kapitel dieser Dissertation dar. Wie in der Einleitung bereits erwähnt, entstammt diese Forschungsarbeit der angewandten Informatik, dementsprechend ist die Umsetzbarkeit eine wichtige Anforderung an das Projekt. Dies wird in diesem Kapitel gezeigt, ein Vorgehen, welches dem Ansatz von Versuch und Irrtum entspricht, der auch im Eingangszitat anklingt.

Die Realisierung eines Programmiermodells erfordert eine geeignete Speicherstruktur für den Modellzustand und Interaktionsmöglichkeiten, über die ein Nutzer das gespeicherte Modell manipulieren kann – diese entsprechen den Sichten im MVC'-Muster. Angelehnt an diese Überlegungen wird in diesem Kapitel zuerst der Entwurf des Modells, dann der der Sichten und zum Schluss der Entwurf des Connectors vorgestellt. Damit sind die Teile der Realisierung des MVC'-Musters beschrieben. Darüber hinaus muss auch eine Abbildung eines konkreten Zustandes des Programmiermodells auf die endgültige Anwendung existieren. Diese wird durch einen Generator und unter Verwendung des in Kapitel 7 beschriebenen Nachrichtenvermittlers realisiert.

8.1 Einleitung

Nachdem in den Abschnitten des vorangegangen Kapitels die Grundlagen für ein Programmiermodell auf Basis von Nachrichten und Komponenten diskutiert wurden, soll nun die Umsetzung dieses Programmiermodells unter Anwendung des in Kapitel 4 diskutierten MVC'-Musters zur Verwendung mit dem Cobamos-Framework vorgestellt werden. Diese Realisierung soll die Anwendbarkeit und Tragfähigkeit zum einen der Konzepte rund um das Cobamos-Framework und das MVC'-Muster und zum anderen des nachrichtenbasierten Programmiermodells belegen. Die hier implementierte Umsetzung ist dabei noch nicht an eine bestimmte Anwendungsdomäne gebunden. Dies geschieht im Rahmen der Anwendung der Ergebnisse dieses Kapitels im folgenden Kapitel 9.

Zunächst wird die Realisierung des Modells diskutiert, da es sich hierbei um den zentralen Bestandteil und die Basis für alle weiteren Überlegungen handelt. Im Anschluss können die Sichten zur Visualisierung und Manipulation des Modells – oder genauer seines Zustandes – entworfen werden. Der zur Integration in die Entwicklungsumgebung notwendige Connector vermittelt die Zugriffe der Sichten auf benötigte Dienste des Cobamos-Frameworks, daher erfolgt seine Realisierung erst nach der der Sichten.

Die in diesem Kapitel vorgestellten Ergebnisse sind noch nicht zur direkten Verwendung durch einen Endnutzer zur Entwicklung von Softwareapplikationen geeignet. Diese Einschränkung resultiert aus dem Hinauszögern der Anpassung der Ergebnisse auf ein konkretes Anwendungsszenario (Beispiele solcher Anpassungen werden in Kapitel 9 gezeigt). Der Entwurf in diesem Kapitel erfolgt so, dass er

auf eine Vielzahl von Anwendungsbeispielen angepasst werden kann. Diese Anpassungen müssen jedoch durch einen Softwareentwickler (dies entspricht der Rolle Softwareentwickler Anwendungsfamilie in Abb. 8.1) vorgenommen werden. Sie bestehen im Wesentlichen in der Bereitstellung von Komponenten, die als Anwendungsbausteine Verwendung finden sollen, und dem Integrieren dieser in das nachrichtenbasierte Programmiermodell (zwei solche Anpassungen werden in Kapitel 9 vorgestellt). Der Endnutzer kann während dieses Entwurfes jedoch nicht außer Acht gelassen werden, da er der spätere Anwender der auf diesem Entwurf basierenden Anwendungen ist. Insbesondere sollen die Sichten auf das Modell von einem Endnutzer bei der Modellierung einer Anwendung verwendet werden. Ansätze zur Unterstützung eines Endnutzers müssen bereits auf dieser Ebene in den Entwurf einfließen.

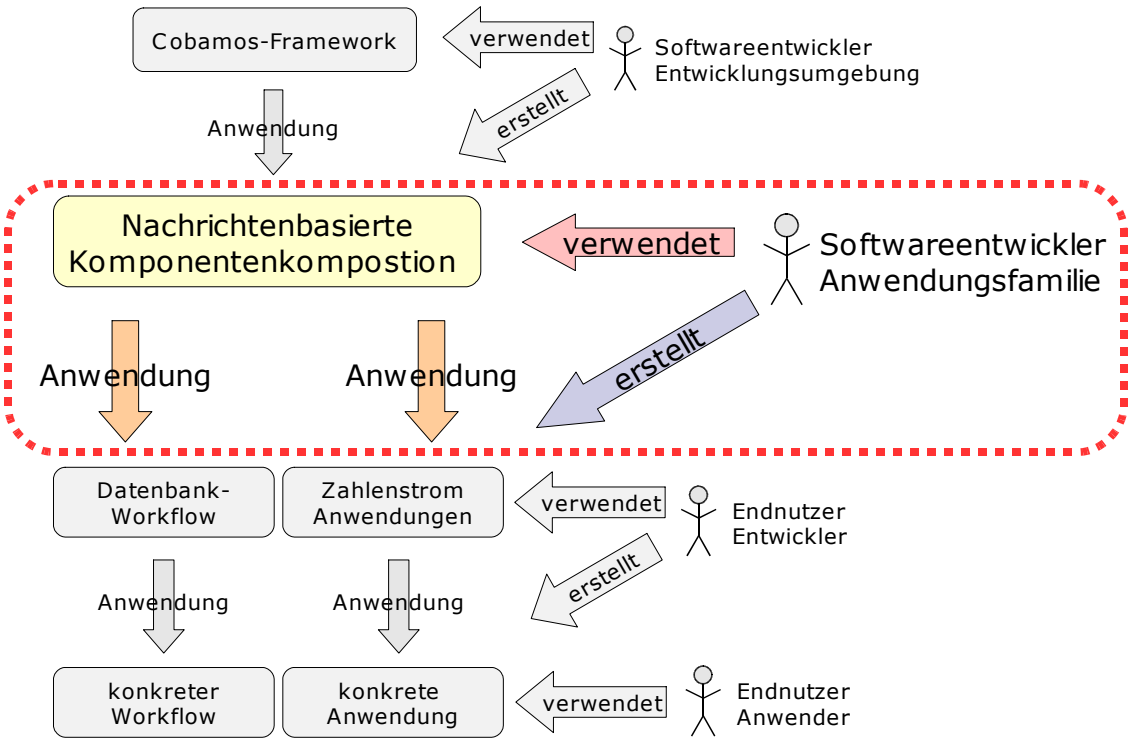


Abb. 8.1: Einordnung der in Kapitel 8 dargestellten Ergebnisse in den Gesamtkontext

Der vorgenommene Entwurf und die Implementation können in diesem Rahmen nicht vollständig und detailliert beschrieben werden. Die Darstellung beschränkt sich daher auf solche Aspekte, die entweder die Umsetzung oder Anwendung der zuvor in dieser Arbeit entwickelten Konzepte verdeutlichen oder die auch in anderen Zusammenhängen von Interesse sein können.

8.2 Das Modell

Das hier zu entwerfende Modell soll in der in den vorangegangenen Kapiteln beschriebenen Entwicklungsumgebung nutzbar sein. Die Anwendungen, die durch einen Endnutzer unter Verwendung dieses Modells erstellt werden sollen, basieren auf Komponenten und dem Austausch von Nachrichten. Hieraus lassen sich bereits erste Anforderungen für den Entwurf ableiten. Es handelt sich um ein Modell im Sinne des MVC'-Musters und muss daher in jedem Fall die Anforderungen, die sich durch die Frameworkarchitektur ergeben, erfüllen. Zum anderen muss es die Speicherung der in einer Applikation eingesetzten Komponenten und der vorhandenen Verbindungen ermöglichen. Dabei ist zu

beachten, dass die alleinige Speicherung des Komponententyps nicht ausreichend ist. Zur Laufzeit der resultierenden Anwendung werden die Nachrichten zwischen Komponenteninstanzen ausgetauscht. In einer Anwendung können durchaus mehrere Instanzen eines Komponententyps existieren; diese müssen unterscheidbar sein, da Instanzen des selben Typs unterschiedliche Verbindungen eingehen können.

Daraus folgt unmittelbar, dass die Verbindungen zwischen Komponenteninstanzen gespeichert werden müssen und nicht zwischen Komponententypen. Allerdings wird durch den Typ einer Komponente festgelegt, welche Verbindungsmöglichkeiten für die Instanzen bestehen. Darüber hinaus bestimmt der Komponententyp eventuelle Konfigurationsmöglichkeiten, die Werte hierfür müssen wiederum instanzweise gespeichert werden. Das Modell muss daher Zugriff auf Beschreibungen der Komponententypen besitzen, die Speicherung der Informationen über das Modell muss an den Instanzen orientiert erfolgen.

Um die einzelnen Instanzen identifizieren zu können, ist ein eindeutiger Bezeichner notwendig. Die vorgenommene Implementation verwendet intern nicht vom Anwender vergebene sondern generierte Bezeichner. Dies soll eine Fehlerquelle ausschließen und den Einsatz dieses Programmiermodells in der Endnutzerprogrammierung erleichtern. Zum Zwecke der Identifikation werden GUIDs verwendet, da diese leicht automatisiert zu erzeugen sind und Eindeutigkeit garantieren. Neben diesem internen Identifikationsmerkmal ist es prinzipiell möglich, weitere Bezeichner für die Komponenteninstanzen zu speichern. Diese können als Anzeigenamen für Komponenteninstanzen und Verbindungen verwendet werden und vom Nutzer eingegeben werden. Dadurch kann die Lesbarkeit von Modellrepräsentationen für einen Anwender verbessert werden.

Die zur Speicherung der Komponenteninstanzen und der Verbindungen verwendeten Datenstrukturen müssen geeignet sein, alle zur Beschreibung einer Anwendung notwendigen Zustände abzubilden. Als Ausgangspunkt für deren Entwurf dient die Visualisierung einer denkbaren Kommunikationsstruktur.

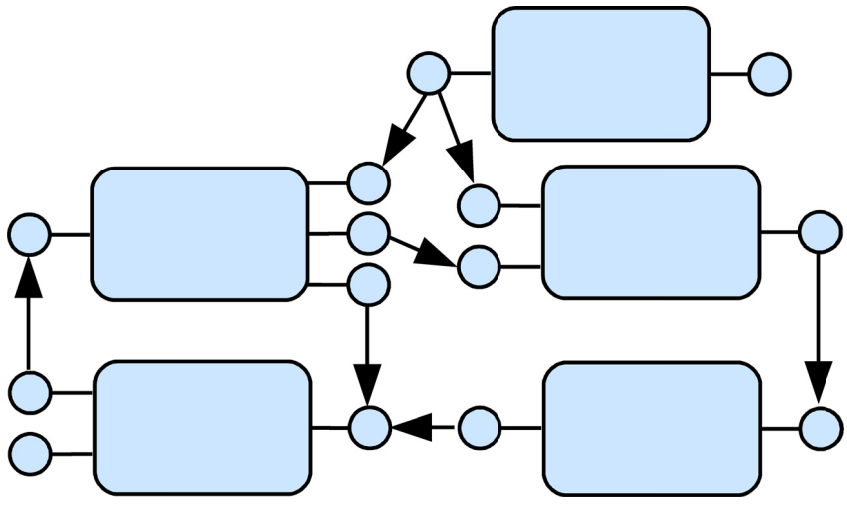


Abb. 8.2: Kommunikationsbeziehungen zwischen Komponenten

Die Abb. 8.2 zeigt einen exemplarischen Kommunikationsgraphen zwischen einigen Komponenteninstanzen. Da noch kein Anwendungsbereich für das Programmiermodell festgelegt werden soll, können auch keine Einschränkungen hinsichtlich der zulässigen Kommunikationsstrukturen gemacht werden. Nicht alle Aus- beziehungsweise Eingänge einer Komponenteninstanz müssen verbunden sein. Zudem erfordern die in Abschnitt 7.3 vorgestellten Kommunikationsmodelle, dass auch mehrere Verbindungen an einem Ein- oder Ausgang existieren dürfen. Zwischen zwei Instanzen können

mehrere Verbindungen unterschiedlicher Art bestehen. Des Weiteren kann der Graph Zyklen enthalten. Dies alles muss beim Entwurf der Datenstrukturen berücksichtigt werden.

Durch die Visualisierung der Kommunikationsstruktur ist es zunächst nahe liegend, den Komponentengraphen direkt als Objektgraphen zu speichern. Ein Objekt entspräche dabei einer Komponenteninstanz, eine Referenz auf ein anderes Objekt einem Nachrichtenkanal. Dieser Ansatz hat insbesondere den Vorteil, dass eine Simulation der späteren Anwendung auf dieser Basis erfolgen kann. Im Verlauf der Realisierung zeigte sich jedoch, dass häufig Zugriffe auf die Gesamtheit der im Modell vorhandenen Instanzen beziehungsweise Verbindungen notwendig sind. Eine solche Abfrage erfordert bei einer Speicherung des Modells in Form eines Graphen eine vollständige Traversierung. Dies muss wie oben angesprochen unter Berücksichtigung möglicher Zyklen erfolgen. Auch Anfragen nach einer bestimmten Komponenteninstanz, wie sie beispielsweise beim Ändern der Eigenschaften vorkommen, erfordern eine Suche im Graphen. Gleiches gilt für den Zugriff auf eine spezielle Verbindung. Hier erweist sich die Verwendung einer Graphstruktur als nachteilig.

Um diese Zugriffe zu beschleunigen, werden nun mit den eindeutigen Bezeichnern der Komponenteninstanzen und der Verbindungen indizierte Hashtabellen verwendet. Es existiert eine Tabelle mit Informationen über die im Modell vorhandenen Komponenteninstanzen und eine weitere mit Informationen über die vorhandenen Verbindungen. Die Zugriffe nach der Gesamtheit der vorhandenen Komponenten beziehungsweise Verbindungen lassen sich auf dieser Basis effizient beantworten. Allerdings erfordert die Suche nach allen Verbindungen, an denen eine bestimmte Komponenteninstanz beteiligt ist – dies ist aufgrund der getrennten Verwaltung von Instanzen und Verbindungen beispielsweise beim Entfernen dieser Instanz aus dem Modell notwendig – einen vollständigen Durchlauf der Verbindungstabelle. Um diese Zugriffe zu beschleunigen, wird ein zusätzlicher Index eingeführt. Dieser enthält zu einem Bezeichner einer Komponenteninstanz die Bezeichner aller ein- und ausgehenden Verbindungen dieser Instanz. Durch die beschriebene lineare Form der Speicherung des Modells entfällt auch die Notwendigkeit einer gesonderten Behandlung von Zyklen im Graphen.

In einer weiteren Entwicklungsstufe dieses Projektes wäre auch eine Kombination aus dem graphbasierten Ansatz und geeigneten zusätzlichen Indizes denkbar, um die Vorteile beider Ansätze zu vereinen. Diese Möglichkeit ist bisher jedoch noch nicht implementiert.

MessageModel		
+ <<Property>>	Connections : ICollection	
+ <<Property>>	Components : ICollection	
+ <<Override>>	AddComponent (ComponentInfo com, string error)	: bool
+ <<Override>>	AddConnection (ConnectionInfo con, string error)	: bool
+ <<Override>>	UpdateComponent (ComponentInfo com, string error)	: bool
+ <<Override>>	RemoveComponent (Guid shapeUID, string error)	: bool
+ <<Override>>	RemoveComponent (ComponentInfo com, string error)	: bool
+ <<Override>>	RemoveConnection (Guid connectionUID, string error)	: bool
+ <<Override>>	RemoveConnection (ConnectionInfo con, string error)	: bool
+ <<Getter>>	get_Connections ()	: ICollection
+ <<Getter>>	get_Components ()	: ICollection
+ <<Override>>	GetComponent (Guid comid)	: ComponentInfo
+ <<Override>>	GetConnection (Guid conid)	: ConnectionInfo
+ <<Override>>	SerializeModel ()	: object
+ <<Override>>	DeserializeModel (object model)	: void

Abb. 8.3: Das Modell (im Sinne des MVC'-Musters)

In Abb. 8.3 ist die Klasse `MessageModel` gezeigt, diese realisiert ein Modell im Sinne des MVC'-Musters und stellt den Datenhintergrund auf Basis der oben besprochenen Datenstrukturen bereit. Dargestellt werden nur die öffentlich zugänglichen Eigenschaften dieser Klasse, Realisierungsdetails werden zugunsten der Übersichtlichkeit unterdrückt.

Die Klasse besitzt öffentliche Methoden zur Manipulation des Modellzustandes. Im Einzelnen sind dies Methoden zum Hinzufügen von Komponenteninstanzen (`AddComponent()`) und Verbindungen (`AddConnection()`) zum Modell. Analog existieren Methoden zum Entfernen der entsprechenden Elemente (`Remove...`). In den Parameterlisten dieser Methoden werden die Typen `ComponentInfo` und `ConnectionInfo` verwendet. Diese kapseln die über eine Komponenteninstanz respektive eine Verbindung benötigten Informationen. Für eine Komponenteninstanz sind dies neben ihrem Bezeichner und den individuellen Werten ihrer Eigenschaften – die Eigenschaften der Komponenten werden eingehender in Abschnitt 8.3 behandelt – einige Informationen hinsichtlich der visuellen Darstellung. Die Informationen zu einer Verbindung sind in der Klasse `ConnectionInfo` gekapselt. Um eine Verbindung zwischen zwei Instanzen eindeutig zu identifizieren, werden die Bezeichner der beiden Instanzen und für jede dieser Instanzen der eindeutige Name des verwendeten Ein- oder Ausgangs gespeichert. Dazu kommt ein eindeutiger Bezeichner für die Verbindung, um ihre Identifikation innerhalb des Modells zu erleichtern.

Für die später (siehe Abschnitt 8.4) beschriebene Realisierung von semantischen Beschränkungen für Verbindungen innerhalb des Modells sind einige Informationen über die Ein- bzw. Ausgänge der an einer Verbindung beteiligten Komponenteninstanzen notwendig, diese werden ebenfalls in der Klasse `ConnectionInfo` abgelegt.

Die Methode `UpdateComponent()` bietet die Möglichkeit, die Eigenschaften einer bereits hinzugefügten Komponente zu ändern. Eine analoge Methode ist für Verbindungen nicht notwendig, da für diese keine zu einem Zeitpunkt nach dem Anlegen konfigurierbaren Eigenschaften verwaltet werden.

Alle Methoden zur Manipulation des Modellzustandes besitzen einen booleschen Rückgabewert, mit dem angezeigt wird, ob die jeweilige Aktion auf dem Modell erfolgreich war. Der Ausgabeparameter `error` enthält bei einer nicht erfolgreich abgeschlossenen Aktion eine Beschreibung des aufgetretenen Fehlers. An dieser Stelle ist anzumerken, dass hier ausschließlich Fehler aufgrund der Modellsemantik[71] gemeint sind. Es ist – dies wird in Abschnitt 8.4 genauer erläutert – möglich, semantische Beschränkungen für zulässige Zustände des Modells zu formulieren. Eine solche Beschränkung kann beispielsweise die Regel sein, dass keine direkt reflexiven Verbindungen einer Komponente zu sich selbst möglich sind oder auch, dass eine Verbindung immer von einem Ausgangs- zu einem Eingangsport verlaufen muss. Da es sich bei einer Verletzung dieser Bedingungen nicht um Fehler im Sinne von Anwendungsfehlern handelt, ist es nicht angebracht, Ausnahmen (Exceptions) zur Kommunikation dieses Fehlerzustandes zu verwenden. Es handelt sich nicht um einen außergewöhnlichen Kontrollfluss aufgrund eines Anwendungsfehlers, sondern um den normalen, vorhergesehenen Kontrollfluss. Die Reaktion auf einen solchen Modellfehler, beispielsweise eine Interaktion mit dem Nutzer, muss in der Sicht erfolgen, welche die fehlgeschlagene Änderung am Modellzustand initiiert hat.

Auch beim Löschen von Modellelementen müssen mögliche Fehler berücksichtigt werden. Zwar könnte beispielsweise der Versuch, ein nicht vorhandenes Element im Modell zu löschen, mit der Begründung ignoriert werden, dass das Modell am Ende der Aktion auf jeden Fall im gewünschten Zustand ist[72]. In diesem Fall könnte also darauf verzichtet werden, einen Fehler zu kommunizieren. Es

[71] Fehler im Sinne von Programmfehlern werden über Ausnahmen kommuniziert.

[72] Dieses sehr pragmatische Vorgehen birgt die Gefahr in sich, Fehler in der Anwendungslogik zu verschleiern.

sind aber Modellbeschränkungen der Form „Es muss mindestens eine Instanz der Komponente X vorhanden sein“ oder „X muss mit Y verbunden sein“ denkbar. Beim Löschen der letzten entsprechenden Instanz beziehungsweise der Verbindung würde ein inkonsistenter Modellzustand entstehen. Da das Modell noch nicht auf konkrete Anwendungsszenarien eingeschränkt werden soll, müssen auch solche Bedingungen durch den Entwurf unterstützt werden. Aus diesem Grund sind die Methoden zum Löschen von Modellelementen entsprechend realisiert.

Über die Eigenschaften `Connections` und `Components` der Klasse `MessageModel` ist der Zugriff auf alle aktuell im Modell vorhandenen Elemente möglich. Die Methoden `GetComponent()` beziehungsweise `GetConnection()` ermöglichen den Zugriff auf eine Komponente respektive eine Verbindung, die Identifikation erfolgt wie oben angesprochen über eine als Parameter übergebene GUID.

Es fällt auf, dass keine der Methoden des in Kapitel 4 besprochenen Interfaces `IModel` in der Darstellung der Klasse `MessageModel` zu sehen sind. Dies ist auf die bereits angesprochene Entwurfsmethodik zurückzuführen, dass für Schnittstellen abstrakte Klassen implementiert wurden, welche soweit möglich Standardimplementationen für die Methoden und Eigenschaften der Schnittstellen bereitstellen. Die Klasse `MessageModel` erbt von der abstrakten Klasse `AbstractMOMModel` und implementiert durch diese Vererbungsbeziehung auch die Schnittstelle `IModel`. Um einen flexiblen Entwurf zu erhalten, sind alle Schnittstellen im Cobamos-Framework derart entworfen, dass sie einzelne Rollen beschreiben, welche die Instanzen der implementierenden Klassen einnehmen können. Die Klasse `MessageModel` ist zum einen ein Modell im Sinne des MVC'-Musters, diese Rolle wird durch das Interface `IModel` beschrieben. Über diese Schnittstelle erfolgt wie in Kapitel 4 erläutert die Einbindung des Modells in die IDE beziehungsweise die Verwaltung desselben. Zum anderen unterstützt die Klasse `MessageModel` die Serialisierung ihrer Daten (vergleiche Abschnitt 6.6), dies wird durch die Implementation des Interfaces `ISerializable` angezeigt. Außerdem ist die Klasse `MessageModel` eine Realisierung eines nachrichtenbasierten Programmiermodells, diese Rolle wird durch die Schnittstelle `IMOMModel` beschrieben. Unter der Annahme, dass nur eine einzige Realisierung für das Modell Verwendung findet, ist diese letztgenannte Schnittstelle – und damit auch die abstrakte Klasse `AbstractMOMModel` – unnötig. Falls jedoch in weiteren Entwicklungsstufen beispielsweise eine Änderung an den internen Datenstrukturen des Modells erfolgen soll, wird diese durch das Interface erleichtert. Da der gesamte Entwurf möglichst flexibel gestaltet sein soll, wurde auch hier die Entscheidung zugunsten der Einführung der Schnittstelle getroffen. Diese Schnittstelle `ISerializable` wird direkt implementiert, da nicht jedes nachrichtenbasierte Modell serialisierbar sein muss. Eine Implementation der Schnittstelle `ISerailizable` durch die abstrakte Klasse `AbstractMOMModel` würde jedoch genau dies ausdrücken.

Zusammen mit der Tatsache, dass in C# keine Mehrfachvererbung existiert, ergibt sich aufgrund der obigen Ausführungen für die Ableitung der Klasse `MessageModel` aus der Schnittstelle `IModel` das in Abb. 8.4 gezeigte Klassendiagramm.

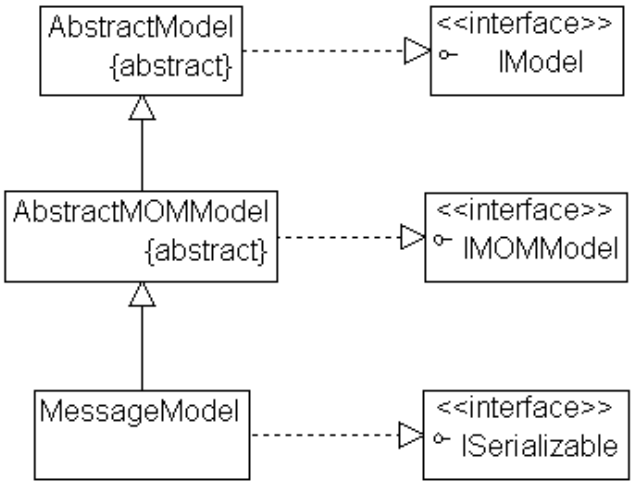


Abb. 8.4: Vererbungshierarchie und Schnittstellen für die Klasse MessageModel

Die Eigenschaften der Komponenten – hierunter sind Funktionalität, Konfigurationsmöglichkeiten und mögliche Verbindungen zusammenzufassen – bestimmen maßgeblich die modellierbare Anwendungsfamilie. Da das hier realisierte Programmiermodell wie bereits angesprochen nicht auf einen konkreten Anwendungsbereich festgelegt sein soll, dürfen diese Komponenteneigenschaften nicht fest im Modell verankert sein. Vielmehr muss eine geeignete Möglichkeit gefunden werden, den hierfür zuständigen Teil der Implementation konfigurierbar zu gestalten. Im nächsten Abschnitt wird der hierfür verwendete Ansatz vorgestellt.

8.3 Die Komponenteneigenschaften

Die Eigenschaften einer Komponente sind ausschlaggebend dafür, wie diese im zu modellierenden System eingesetzt werden kann. Diese Eigenschaften lassen sich in mehrere Kategorien unterteilen. Zum einen gibt es Eigenschaften, welche die Konfiguration einer Komponenteninstanz verändern und durch den Endanwender bei der Modellierung einer Softwareanwendung manipulierbar sein sollen. Daneben gibt es Komponenteneigenschaften, die von der Entwicklungsumgebung zum Generieren der Zielanwendung benötigt werden. Hierzu zählen Informationen über die möglichen Ein- und Ausgänge der Komponenten, die physikalische Datei, in der die Komponente realisiert ist und Ähnliches. Diese Eigenschaften dürfen durch den Endanwender in der Regel nicht verändert werden[73] und müssen mit den Gegebenheiten der beschriebenen physischen Komponente übereinstimmen. Daneben sind noch Initialisierungsanweisungen für die Komponenten denkbar, welche nicht von der Entwicklungsumgebung benötigt werden, aber ebenfalls nicht vom Endanwender manipuliert werden dürfen.

Um auch zum Zeitpunkt dieses Entwurfs noch nicht festgelegte Eigenschaften verarbeiten zu können, wird eine – auch an den Implementationsnotwendigkeiten orientierte – Sprache zur Beschreibung der Komponenteneigenschaften verwendet. Diese Beschreibungssprache ist XML-basiert. Die Wahl dieses Formates stellt dabei keine Einschränkung des Entwurfs für das Programmiermodell dar. Die zur Verarbeitung der Komponentenbeschreibungen beziehungsweise zum Ermitteln der Eigenschaften benötigten Klassen sind in Form einer Komponente gekapselt und daher austauschbar. Die

[73] Bei Anwendern mit entsprechendem Kenntnisstand ist diese Forderung nicht notwendig. Denkbar sind dann Anwendungsszenarien, in denen ein Anwender unterschiedliche Versionen einer Komponente durch Angabe der zu verwendenden physischen Datei auswählen kann.

Schnittstellen, über welche die Eigenschaften von der Entwicklungsumgebung abgerufen werden, sind unabhängig von der Form der Beschreibung. Die Lesbarkeit der XML-Beschreibung erleichtert die Entwicklung erheblich, dies war ein wesentlicher Grund für die Verwendung dieses Formates. Zudem existiert eine Vielzahl von Werkzeugen und Bibliotheken zum Umgang mit XML, auch dies ist ein Vorteil im Entwurf.

Auch die im weiteren Entwurf verwendete Form der Speicherung der Komponentenbeschreibungen in einer Datei ist keine notwendige Bedingung, die Daten wurden während der Entwicklung prototypisch sowohl in XML-Dateien gespeichert als auch in einer über einen Webservice angebundenen relationalen Datenbank. Die Verwendung des Webservices erlaubt eine zentrale Bereitstellung der Komponentenbeschreibungen, dies kann zu einer Verringerung des Wartungsaufwands führen.

Der entwickelte Ansatz zur Beschreibung der Komponenteneigenschaften soll im Folgenden am Beispiel einer Komponente vorgestellt werden, welche aus dem bereits angesprochenen Anwendungsbeispiel der Softwareappliaktionen zur Manipulation von Datenströmen aus ganzen Zahlen stammt. Die beschriebene Komponente stellt Funktionalität zur Erzeugung von Folgen aus ganzen Zahlen bereit und wird im Weiteren als Generator bezeichnet. Die unten stehende Abb. 8.5 zeigt die Interaktionsoberfläche einer Instanz dieses Generators. Neben statischen Zahlenfolgen und zufälligen Zahlenfolgen aus einem bestimmten Intervall können auch Zahlenfolgen aufgrund einer als Parameter bei der Instanzierung der Komponente übergebenen Formel erzeugt werden. Diese Formel ist eine durch den Nutzer änderbare Eigenschaft, welche vor der Instanzierung der Komponente festgelegt sein muss. Die Auswahl des Erzeugungsverfahrens und die Frequenz können zur Laufzeit geändert werden, das Textfeld am unteren Rand der Interaktionsoberfläche zeigt die gerade erzeugte Zahl an.

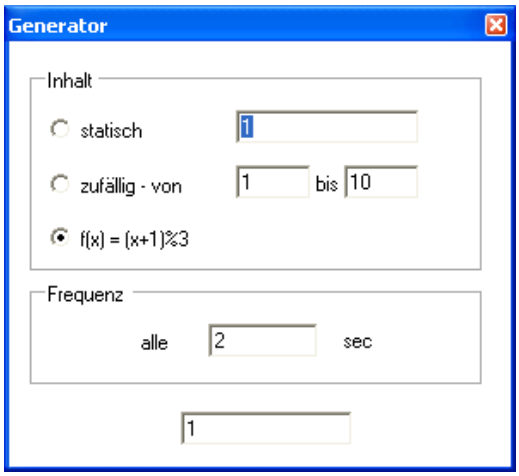


Abb. 8.5: Die Interaktionsoberfläche einer Komponenteninstanz vom Typ Generator

Die Beschreibung der Eigenschaften richtet sich nach den Implementationsnotwendigkeiten, da hier die Informationen bereitgestellt werden müssen, die für den Umgang mit den Komponenteneigenschaften innerhalb der Realisierung notwendig sind. Mit Veröffentlichung des .NET-Frameworks hat die Firma Microsoft auch ein sehr flexibel nutzbares Steuerelement zur Realisierung von Konfigurationsoberflächen, das PropertyGrid [Mic06k], freigegeben. Dieses zeigt einen Schnappschuss der Eigenschaftswerte eines übergebenen Objektes an. Auf dieser Komponente basiert auch der beispielsweise in Abb. 8.6 zu sehende Eigenschaftseditor. Sie wird jedoch nicht standardmäßig bei den verfügbaren Oberflächenelementen in der Microsoft Visual Studio-Entwicklungsumgebung angezeigt. Mit dem Interface `ICustomTypeDescriptor` des Namensraumes `System.ComponentModel` [Mic06l] stellt

Microsoft die Basis bereit, um selbst Informationen zu einer Instanz festlegen zu können, die anders als im Normalfall nicht auf Basis der Metadaten des Objektes ermittelt werden. Dies erlaubt es prinzipiell, dynamisch Eigenschaften für Objekte festzulegen und entspricht damit genau den Anforderungen, die sich in diesem Entwurf aufgrund der zur Entwurfszeit nicht bekannten Komponenteneigenschaften ergeben. Allerdings sind diese Möglichkeiten, flexibel Eigenschaften zu beschreiben, nur wenig dokumentiert. In [All02] werden einige Klassen zur Verfügung gestellt, die diese Möglichkeiten leichter zugänglich machen, diese wurden auch bei der Implementation der internen Beschreibung der Eigenschaften verwendet.

```
(1)  <?xml version="1.0" encoding="UTF-8"?>
(2)  <entities xmlns="http://tempuri.org/PropertyFile1.xsd">
(3)  <component  type="Generator"
(4)              shape="Cobamos.MOM.MOMShapes.Generator"
(5)              shapeKey="{8A28F072-7A38-4dc7-8381-F41D0D8D787B}">
(6)    <property name="Name"
(7)              description="Name einer Komponenteninstanz"
(8)              category="Instanzinformation"
(9)              readonly="0"
(10)             type="System.String">
(11)   </property>
(12)   <property name="Formel"
(13)             description="Nach dieser Formel wird der neue Wert aus dem
(14)                          vorhergehenden berechnet. Die Variable x bezeichnet
(15)                          den alten Wert. Eine Formel der Form (x+1)%3 erzeugt
(16)                          eine Zahlenfolge der Form 0,1,2,0,1,2,.."
(17)             category="Instanzinformation"
(18)             readonly="0"
(19)             type="System.String"
(20)             typeeditor="Cobamos.MOM.Editoren.FormelEditor">x+1
(21)   </property>
(22)   <property name="Startwert"
(23)             description="Der Startwert für die Berechnugen der Werte. Der erste
(24)                          gesendete Wert ist f(Startwert)."
(25)             category="Instanzinformation"
(26)             readonly="0"
(27)             type="System.String">0
(28)   </property>
(29)   <property name="Executable"
(30)             description="Die Komponentendatei."
(31)             category="Laufzeit"
(32)             readonly="1"
(33)             type="System.String">Generator.exe
(34)   </property>
(35)   <port name="Out" type="OUT">integer</port>
(36)   <initialization></initialization>
(37) </component>
(38) </entities>
```

Codebeispiel 8.1: Beschreibung einer Komponente zur Erzeugung von Zahlenfolgen

In Codebeispiel 8.1 ist die Beschreibung der Generatorkomponente (genauer des Komponententyps) gezeigt. Die Zeilen 1 und 2 beinhalten die XML-Deklaration und das Wurzelelement für Beschreibungsdokumente mit der Festlegung eines Schemadokumentes. Die Beschreibung der Komponente beginnt ab Zeile 3 mit dem Element `component`, welches eine Komponentenbeschreibung kenn-

zeichnet. Das Attribut `type` gibt den Typ der Komponente an, die Attribute `shape` und `shapeKey` werden für die Darstellung der Komponenten in einer der Sichten (der Graphdarstellung) benötigt. Sie verweisen auf Klassen, welche eine visuelle Repräsentation der Komponente implementieren. Die Typangabe wird zur Identifikation der Komponente verwendet. Sie ist unabhängig von der Angabe der physischen Datei, in welcher die Komponente realisiert ist.

Es ist nicht möglich, die denkbaren Eigenschaften für alle möglichen Komponenten zu diesem Zeitpunkt festzulegen. Daher basiert der hier gezeigte Ansatz auf der Beschreibung der Eigenschaften, dabei müssen die für ihre Verwendung in der Entwicklungsumgebung notwendigen Informationen bereitgestellt werden. Eine Eigenschaft – als Beispiel dient hier die Eigenschaft `Name`, welche dazu dient einen menschenlesbaren Bezeichner für eine Komponenteninstanz vergeben zu können – wird durch ein Element vom Typ `property` beschrieben (Zeile 6). Sie hat einen Namen, welcher durch den Wert des Attributs `name` bestimmt wird (Zeile 6, im Beispiel `Name`). Dieser muss die Eigenschaft innerhalb eines Komponententyps eindeutig identifizieren. Neben dem Bezeichner ist der Typ der Eigenschaft von Bedeutung, dieser wird durch den Wert des Attributs `type` in Zeile 10 bestimmt (der Bezeichner einer Komponenteninstanz ist eine Zeichenkette, hier wird also der entsprechende Datentyp der Programmiersprache C# `System.String` verwendet). Ein eventueller Vorgabewert für eine Eigenschaft kann als Inhalt in dem Element `property` gespeichert werden (dies ist für den Instanzbezeichner nicht sinnvoll).

Die übrigen Informationen werden für die visuelle Aufbereitung der Komponenteneigenschaften benötigt. Die Beschreibung gibt dem Anwender zusätzliche Informationen über diese Eigenschaft, die Kategorien ermöglichen eine Gliederung der Eigenschaften. Mit dem Attribut `readonly` (siehe Zeile 9) wird festgelegt, ob eine Änderung des vorgegebenen Wertes durch den Anwender möglich ist oder nicht. Durch das Attribut `category` ist eine Einteilung der Eigenschaften bei der Darstellung in unterschiedliche Kategorien möglich. Eine Beschreibung der Eigenschaft – diese wird dem Anwender bei der Verwendung des Eigenschaftseditors angezeigt – kann mit dem Attribut `description` festgelegt werden.

Über die Angabe eines Typeditors ist es möglich, dass ein spezieller Editor für eine Eigenschaft bereitgestellt wird. Dies kann bei einigen Eigenschaften die Eingabe eines Wertes für den Anwender erheblich erleichtern. Ein Beispiel hierfür sind die Schrift- und Farbauswahldialoge, wie sie in aktuellen Softwareapplikationen Verwendung finden. Die Festlegung eines solchen Editors erfolgt mit dem Wert des optionalen Attributs `typeeditor`, siehe die Beschreibung der Eigenschaft `Formel` in Zeile 20, der Editor ist in Abb. 8.6 zu sehen.

Für alle Komponenten, die später in eine Anwendung integrierbar sein sollen, müssen ihre Verbindungsmöglichkeiten beschrieben werden. Im nachrichtenbasierten Modell können diese Verbindungsmöglichkeiten einer Komponente wie erwähnt als Ports gesehen werden. Ein Port (Element `port`) besitzt neben einem Bezeichner (festgelegt durch den Wert des Attribut `name`) – dieser muss innerhalb der Komponente eindeutig sein, da er zur Identifikation des Ports verwendet wird – eine Orientierung (diese wird als Wert des Attributs `type` angegeben). Es wird zwischen Ein- und Ausgangsports unterschieden. Dabei gibt es keine Festlegung, dass eine Komponente sowohl über Ein- als auch über Ausgänge verfügen muss. Auch reine Nachrichtenquellen beziehungsweise -senken sind möglich. Die Daten im Inneren eines Portelementes sind reserviert, um festlegen zu können, welches Vokabular der beschriebene Port verstehen kann bzw. welche Struktur die Nachrichten aufweisen können. Diese Informationen werden bei der Umsetzung von semantischen Beschränkungen für das Modell in Abschnitt 8.4 benötigt. Der Generator verfügt nur über einen Ausgangsport, dessen Beschreibung ist in Zeile 35 zu sehen.

Um zusätzliche Eigenschaften, die eine Komponente benötigt, die aber weder dem Endanwender zur Verfügung stehen sollen noch von der Cobamos-Anwendung verwendet werden, in diese Beschreibung zu integrieren, wird ein einfaches Element (das Element `initialization` in Zeile 36) eingeführt. Der Inhalt dieses Elementes – das gesamte XML-Fragment – wird an die Komponente zum Zeitpunkt der Initialisierung übergeben. Da diese Eigenschaften keine Veränderungen innerhalb der Entwicklungsumgebung erfahren können, sind sie für alle Komponenteninstanzen gleich, die Speicherung an dieser einen Stelle ist also ausreichend. Ein Anwendungsbeispiel ist die Bereitstellung von Konfigurationsdaten für die Komponenten, beispielsweise der Pfad zu einer Datenbank zum Ablegen von Protokolldaten.

Die Manipulation der nutzereditierbaren Eigenschaften der Komponenteninstanzen kann zur Laufzeit der Entwicklungsumgebung über verschiedene Sichten erfolgen. Im Vorgriff auf den Abschnitt 8.5, in welchem die realisierten Sichten für das nachrichtenbasierte Programmiermodell vorgestellt werden, zeigt die Abb. 8.6 den implementierten Eigenschaftseditor. Diese Sicht auf das Modell ist – wie die übrigen in Abschnitt 8.5 vorgestellten auch – für die Nutzung durch einen Endnutzer beim Modellieren einer Anwendung gedacht. Eine Sicht auf das Modell im Sinne des MVC'-Musters muss nicht das gesamte Modell darstellen. Eine Beschränkung auf bestimmte Modellaspekte kann diese betonen und das Verständnis beziehungsweise die Interaktion mit dem Modell für einen Anwender erleichtern. In diesem Sinn ist der Eigenschaftseditor als Sicht auf einen Modellausschnitt realisiert. Dargestellt werden jeweils nur die Eigenschaften einer selektierten Komponenteninstanz. Im gezeigten Beispiel ist eine Instanz vom Typ Generator ausgewählt, um den Zusammenhang mit der in Codebeispiel 8.1 gezeigten Komponentenbeschreibung zu verdeutlichen. Die Werte der Attribute mit der Bezeichnung `category` in der Beschreibung bestimmen die Einordnung der Eigenschaften in die Kategorien („Instanzinformation“ und „Laufzeit“) in der Darstellung im Eigenschaftseditor.

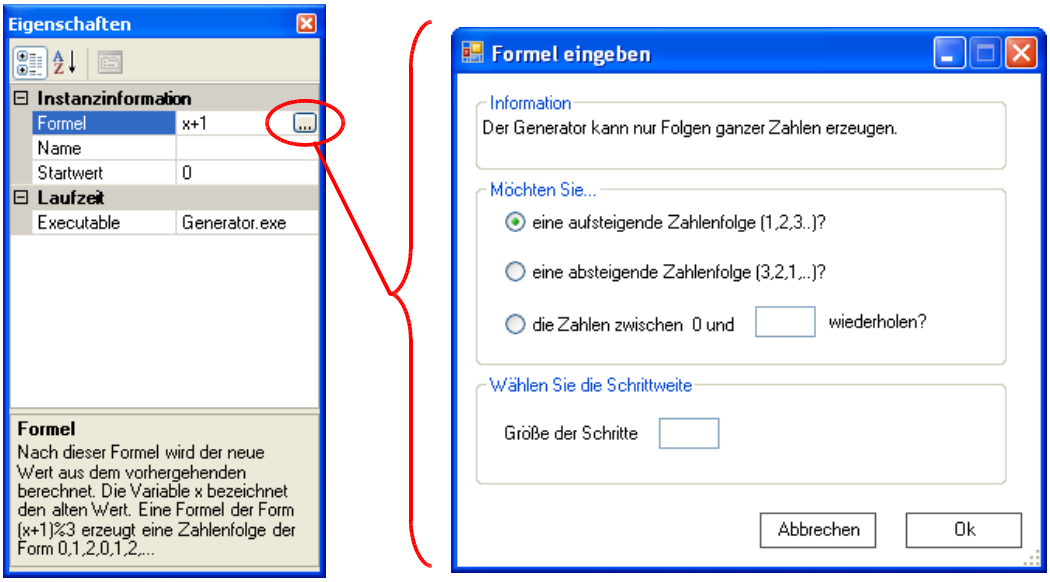


Abb. 8.6: Anzeige der Eigenschaften einer Komponenteninstanz vom Typ Generator

Darüber hinaus ist in Abb. 8.6 im unteren Bereich des linken Fensters die Anzeige des Beschreibungstextes zu sehen, der für die Eigenschaft Formel durch das Attribut `description` festgelegt wurde. Die Schaltfläche, welche am Ende der Zeile zur Eingabe der Formel zu sehen ist, öffnet den durch das Attribut `propertyeditor` bestimmten Editor für diese Eigenschaft. Solche Editoren müssen nicht bereitgestellt werden, ohne die Angabe erfolgt die Werteeingabe im

entsprechenden Feld des Eigenschaftseditors. Spezialisierte Editoren können die Eingabe der Werte für einen Anwender jedoch erleichtern, gerade im Bereich der Endnutzerprogrammierung sind hier Vorteile zu erwarten. Denkbar sind beispielsweise zusätzliche Hilfen bei der Werteeingabe oder auch ein dialogbasiertes Abfragen von Werten für bestimmte Anwendergruppen. In der Abb. 8.6 ist auf der rechten Seite ein exemplarisch implementierter Editor zur Unterstützung bei der Eingabe der Erzeugungsformel für den Generator dargestellt.

Der hier gezeigte Ansatz zur Beschreibung der Eigenschaften einer Komponente ist eine Alternative zu dem in Abschnitt 4.3.2 vorgestellten Mechanismus der Reflexion. Während bei der Reflexion die Eigenschaften einer Komponente aus dieser selbst ermittelt werden und daher untrennbar mit dieser verbunden sind, handelt es sich bei der Verwendung der XML-Dokumente um eine externe und von der Komponente trennbare Beschreibung. Aus der Realisierung der dynamischen Interaktionskomponenten in Abschnitt 4.3.5 ergibt sich, dass ein reflexiver Ansatz sich nicht notwendigerweise auf die Ermittlung von Metainformationen beschränken muss. Bei dem dort gezeigten Vorgehen kann eine Komponenteninstanz nach bestimmten Informationen – dort handelte es sich um Informationen bezüglich der Integration in die visuelle Oberfläche der Entwicklungsumgebung – befragt werden.

Es wäre prinzipiell denkbar, auch hier in den Komponenten, die als Bausteine der Anwendungsfamilien dienen sollen, die notwendigen Informationen zu integrieren und dadurch auf die XML-Dokumente zu verzichten. Allerdings müssten auch in diesem Fall die Eigenschaften durch Metainformationen beschrieben werden. Zudem ist die Möglichkeit, die Beschreibung von den Komponenten zu trennen, in diesem Anwendungsszenario von Vorteil und eröffnet eine Reihe interessanter Möglichkeiten. Da die Entwicklungsumgebung in der aktuellen Version keinen Zugriff auf die eigentlichen Komponenten benötigt, lassen sich beispielsweise auch Anwendungen realisieren, bei denen durch die IDE nur eine Beschreibung einer Anwendung generiert wird. Die eigentliche Komposition der Anwendung unter Verwendung der Komponenten kann durch eine separate Softwareapplikation erfolgen. Diese kann beispielsweise auf einem zentralen Server ausgeführt werden. Hieraus resultiert als Vorteil eine leichtere Wartung der Komponenten, da diese bei einem Versionswechsel nur an einer Stelle getauscht werden müssen. Außerdem ist so eine zentrale Kontrolle über den Generierungsprozess möglich, beispielsweise um entsprechende Abrechnungsmodelle zu realisieren. Ein Kunde würde in einem solchen Szenario eine Anwendung mittels der IDE beschreiben und diese Beschreibung auf den „Generierungsserver“ eines Herstellers übertragen. Dem Kunden werden dann die für die Erzeugung der Anwendung benötigten Komponenten in Rechnung gestellt. Ein weiteres Einsatzfeld, in dem eine solche zentrale Lösung von Interesse wäre, sind Schulungs- und Prüfungssituationen. Die erzeugte Anwendungsbeschreibung wird dann nicht an einen Generator, sondern an ein Prüfprogramm übergeben. Neben diesen Einsatzfeldern, bei denen die mögliche Zentralisierung im Vordergrund steht, ist die Verwendung einer separaten Beschreibung auch die Grundlage für die im Folgenden aufgezeigte Möglichkeit, unter gewissen Bedingungen die Bausteine einer Anwendung unter Beibehaltung des Steuerflusses zwischen den Komponenten austauschen zu können.

8.4 Realisierung von Bedingungen und Modellbeschränkungen

Das in diesem Kapitel entworfene Programmiermodell soll innerhalb einer auf dem Cobamos-Framework basierenden Entwicklungsumgebung Anwendung finden, dies ist bei der Realisierung entsprechend zu berücksichtigen. Wie in Kapitel 4 beim Entwurf des Frameworks bereits angesprochen, werden Beschränkungen, die für das Programmiermodell gelten, durch die Realisierung des Modells geprüft und nicht durch jede einzelne Sicht. Die Sichten sollten nur die für sie

spezifischen Beschränkungen überprüfen. Im Falle einer Sicht, die eine textuelle Notation verwendet, wäre dies beispielsweise die syntaktische Korrektheit der eingegebenen Anweisungen.

Das nachrichtenbasierte Programmiermodell ist, so wie es in Kapitel 7 entworfen wurde und hier realisiert wird, nicht auf eine Anwendungsdomäne festgelegt. Eine solche Festlegung bestimmt die modellierbare Anwendungsfamilie und beinhaltet die konkrete Definition von zur Verfügung stehenden Komponenten. Darüber hinaus ist eine Notation vonnöten, mit der Exemplare der Anwendungsfamilie beschrieben werden können. Die Festlegung geeigneter Notationen wird in Abschnitt 8.5 behandelt. Es ist jedoch wahrscheinlich, dass nicht jede beliebige Kombination von Komponenten sinnvoll ist. Daher muss es Möglichkeiten geben, entsprechende Beschränkungen zu formulieren. Die Notation zusammen mit den Beschränkungen entspricht der Festlegung einer domänenspezifischen Sprache[74].

Bei der Entwicklung einer domänenspezifischen Sprache auf Basis des nachrichtenbasierten Programmiermodells können zwei Arten von Beschränkungen unterschieden werden. So existieren neben den Beschränkungen aufgrund des Programmiermodells – beispielsweise dass Nachrichtenkanäle von einem Aus- zu einem Eingang verlaufen müssen oder dass immer beide Enden einer Verbindung mit einer Instanz verbunden sein müssen – auch durch die Anwendungsdomäne vorgegebene. Eine solche könnte beispielsweise die zulässige Reihenfolge von Komponenten zur Beschreibung eines Workflows betreffen. Aber auch die Festlegung, dass bestimmte Komponenten keine reflexiven Beziehungen besitzen dürfen, fällt in diese Gruppe. Die domänenspezifischen Einschränkungen können erst bei der Realisierung einer konkreten Sprache, also bei der Anwendung des Programmiermodells auf eine konkrete Anwendungsdomäne, festgelegt werden. Der für die Überprüfung solcher Regeln zuständige Teil des Programmiermodells muss daher entsprechend konfigurierbar sein.

Bei dem für diese Überprüfungen relevanten Teil des Entwurfs ist zu beachten, dass bei der Realisierung einer Applikation unter Verwendung dieses Programmiermodells zwei Sprachen eine Rolle spielen. Dies ist zum einen die Sprache, auf der die zwischen den Komponenten ausgetauschten Nachrichten basieren und zum anderen die Sprache, mit der der Endanwender die Zielapplikation beschreibt.

Die Sprache der Nachrichten ist dafür maßgeblich, ob eine Verbindung zwischen einem Komponentenausgang und einem Komponenteneingang grundsätzlich möglich ist. Die Ein- beziehungsweise Ausgänge einer Komponente können unterschiedliche Sprachen unterstützen, es ist beispielsweise eine Komponente mit einem Steuer- und einem Datenport denkbar. Die Festlegung der Sprachen kann daher nicht für eine Komponente insgesamt erfolgen sondern muss auf der Ebene der Ports geschehen.

Diese Festlegung der von einem Port unterstützten Sprache erfolgt – wie bei der Diskussion von Codebeispiel 8.1 auf Seite 139 angesprochen – in der Komponentenbeschreibung. Die dort gezeigte Festlegung auf Nachrichten, welche aus ganzen Zahlen bestehen, ist vergleichsweise einfach. Für die Familie der zahlenstrombasierten Anwendungen (vergleiche Kapitel 9), zu der diese Beschreibung gehört, ist sie jedoch ausreichend. Da es aber auch möglich sein soll, komplexere Nachrichtensprachen zu beschreiben, ist in der XML-Beschreibung der Inhalt des Elementes `port` für diesen Zweck reserviert. Dies ermöglicht es prinzipiell, auch aufwendigere XML-Strukturen zur Beschreibung der von einem Port unterstützten Sprache zu verwenden. Es ist sogar denkbar, binäre Nachrichtenformate

[74] In Appendix A werden die Begriffe Anwendungsfamilie, Domäne und domänenspezifische Sprache im Rahmen eines Exkurses detaillierter betrachtet.

in einer geeigneten textuellen Kodierung (beispielsweise auf Basis der Base64-Kodierung [Jos03]) anzugeben.

Die Informationen über einen Ein- oder Ausgang – also sein Name, Sprache und die Richtung[75] – sind für alle Instanzen eines Komponententyps identisch. Daher ist es auch nicht notwendig, diese bei den Informationen über die Komponenteninstanzen – hierfür ist die weiter oben bereits angesprochene Klasse `ComponentInfo` verantwortlich – abzulegen, da dies zu einer unnötigen wiederholten Speicherung dieser Daten führen würde. Die durch das Element `port` in der Komponentenbeschreibung festgelegten Informationen werden in der Klasse `PortInfo` (siehe Abb. 8.7) gekapselt. Die Eigenschaft `PortType` entspricht der Richtungsangabe, die Eigenschaft `Name` enthält den Bezeichner des Ports und die Eigenschaft `Language` dient der Beschreibung der Sprache des Ports. Auf die Realisierung dieser Sprachbeschreibung respektive der Prüfung auf Kompatibilität zweier Ports wird nun genauer eingegangen.

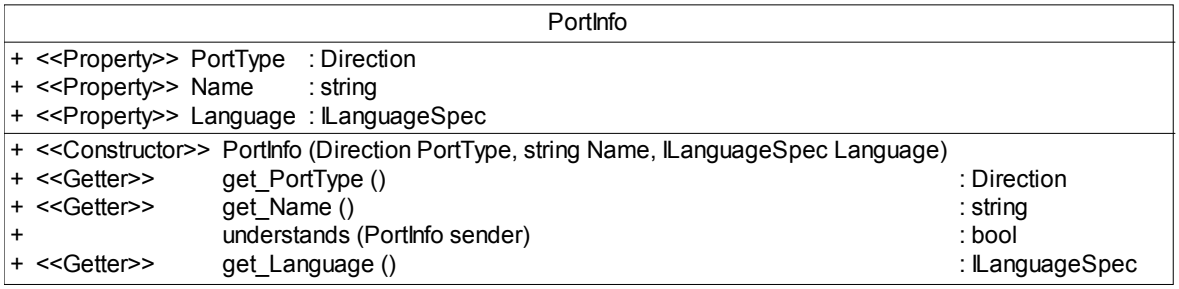


Abb. 8.7: Die Klasse PortInfo

Beim Anlegen einer Verbindung zwischen zwei Komponenteninstanzen im Modell muss die Kompatibilität der beteiligten Ports geprüft werden. Ein denkbarer Entwurf hierfür ist es, die notwendige Methode in der Klasse anzusiedeln, die das Modell realisiert. Diese Methode ist dann dafür verantwortlich, die in den Instanzen der Klasse `PortInfo` gekapselten Informationen über die unterstützte Sprache eines Komponentenports auszuwerten und zu vergleichen. Bei genauerer Betrachtung ist die Kompatibilität mit einem anderen Port jedoch eine Eigenschaft eines Ports. Die Prüfung muss in beide Richtungen – also beispielsweise sowohl `PortA.istKompatibel(PortB)` als auch `PortB.istKompatibel(PortA)` – erfolgen können, je nachdem ob aus Sicht des Senders oder des Empfängers der Nachrichten – oder allgemeiner des Ein- beziehungsweise Ausgangs der beteiligten Instanzen – geprüft wird. Unter diesem Gesichtspunkt ist die Kompatibilitätsprüfung symmetrisch, dem wird durch das Ansiedeln der hierfür notwendigen Funktionalität zur Kompatibilitätsprüfung der Klasse `PortInfo` Rechnung getragen. Das Ergebnis der Kompatibilitätsprüfung ist immer abhängig von der Richtung des Nachrichten- oder Datentransfers. Da die Ports jedoch explizit ihre Richtung als Eigenschaft besitzen und in dem vorgeschlagenen Modell keine bidirektionalen Ports existieren, kann dies bei der Implementation einer konkreten Prüfung dementsprechend berücksichtigt werden.

Um diese Prüfung auf Sprachkompatibilität der Ports austauschbar zu gestalten, wird das Interface `ILanguageSpec` (siehe Abb. 8.8) eingeführt. Prinzipiell ist es möglich, dass in dem Port-Element einer

[75] Der Begriff Richtung oder Orientierung im Zusammenhang mit den Ports einer Komponente wird in dieser Arbeit immer aus Sicht der Komponente definiert. Ein Eingang dient also dazu, der Komponente Daten oder Nachrichten zu übergeben. Es ist wichtig zu bemerken, dass sich diese Unterscheidung nicht auf die Realisierung einer Schnittstelle oder die Richtung von Methodenzugriffen bezieht, sondern auf die Richtung des Datentransfers. Im Fall der Realisierung in Form eines objektorientierten Interfaces können sowohl Komponentenein- als auch -ausgang durch den Aufruf von Methoden eines Interfaces realisiert sein.

Komponentenbeschreibung neben der Sprachbeschreibung für den Port auch eine Komponente angegeben wird, welche die zu verwendende Realisierung einer Prüffunktionalität bereitstellt. Dadurch wird es möglich, unterschiedliche Verfahren für die Überprüfungen der Kompatibilität der Nachrichtensprache innerhalb eines Modells – und auch innerhalb einer Komponente – zu verwenden.

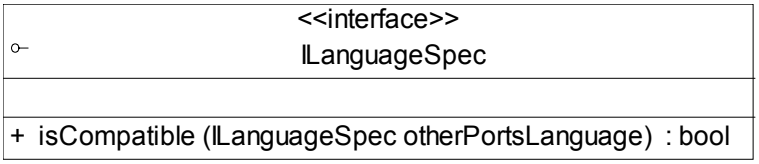


Abb. 8.8: Das Interface ILanguageSpec

Die Erzeugung der Instanzen von Klassen, die das Interfaces `ILanguageSpec` realisieren, kann nicht direkt erfolgen, da die Bezeichner dieser Klassen zum Entwurfszeitpunkt nicht bekannt sind. Die Instanzierung erfolgt daher unter Verwendung der Klasse `Activator` des Namensraumes `System` der C#-Klassenbibliothek. Im Gegensatz zur in Codebeispiel 6.2 gezeigten Anwendung wird hier jedoch ein parametrisierter Konstruktor verwendet. Die notwendige Funktionalität ist in einer Erzeugermethode gekapselt, welche den Typ der zu erzeugenden Instanz und die Parameter für den Konstruktor übergeben bekommt.

Im Folgenden soll verdeutlicht werden, wie eine Kompatibilitätsprüfung bezüglich der Nachrichtensprache realisiert werden kann. Zu diesem Zweck wird die Implementation einer konkreten Prüffunktionalität für die Verwendung mit den in Codebeispiel 8.1 gezeigten Beschreibungen besprochen. Diese ist in Codebeispiel 8.2 zu sehen. Der Konstruktor der Klasse `SimpleLanguageSpec` bekommt den gesamten Inhalt des XML-Elementes `port` aus der Beschreibung der Komponente übergeben. Diese Konvention hinsichtlich der Parameter des Konstruktors kann nicht durch die Schnittstelle festgelegt werden. In dem hier betrachteten einfachen Fall wird nur auf Typgleichheit der Sprachbeschreibungen (Zeile 14) und auf Gleichheit des Sprachbezeichners (Zeile 17 und 18) geprüft, komplexere Kompatibilitätsprüfungen könnten beispielsweise Teilmengeneigenschaften der Sprachen überprüfen. Ausgeführt wird eine solche Kompatibilitätsprüfung im Rahmen der im Folgenden vorgestellten Prüfungen von Modellbeschränkungen beim Anlegen einer Verbindung.

```
(1)   public class SimpleLanguageSpec :ILanguageSpec
(2)   {
(3)      private string _portText;
(4)      public string PortText
(5)      {
(6)         get { return _portText; }
(7)      }
(8)      public SimpleLanguageSpec(string portText)
(9)      {
(10)        this._portText = portText;
(11)     }
(12)     public bool isCompatible(ILanguageSpec otherPortsLanguage)
(13)     {
(14)        if (!this.GetType().Equals(otherPortsLanguage.GetType()))
(15)           return false;
(16)        else
(17)           return this.PortText.Equals(
(18)                  (otherPortsLanguage as SimpleLanguageSpec).PortText);
(19)     }
(20)  }
```

Codebeispiel 8.2: Die Klasse SimpleLanguageSpec

Neben den Prüfungen hinsichtlich der Sprache der Komponentenports müssen auch Möglichkeiten zur Prüfung der übrigen eingangs erwähnten Bedingungen geschaffen werden. In der derzeitigen prototypischen Realisierung wird dies durch austauschbare Prüfkomponenten umgesetzt. Um eine Schnittstelle für diese festzulegen, muss zunächst ermittelt werden, welche Arten von Prüfungen notwendig sein können und wie ein Zugriff auf die notwendige Funktionalität geschehen kann. Grundsätzlich können sich Prüfungen in diesem Zusammenhang auf die Zulässigkeit einer Beziehung zwischen zwei Komponenteninstanzen, auf das Vorhandensein oder den Zustand einer Komponenteninstanz und auf das gesamte Modell beziehen. Wie bereits beim Entwurf der Schnittstelle für das Modell angesprochen, sollten Ausnahmen für die Fehlerbehandlung verwendet werden und nicht für den normalen Kontrollfluss der Anwendung (vergleiche Seite 135, Abschnitt 8.2). Aus diesem Grund liefern die Methoden zur Prüfung einen booleschen Wert zurück, welcher ein Bestehen oder nicht Bestehen der Überprüfung anzeigt. Zudem wird ein Parameter `error` deklariert, der zur Ausgabe einer Fehlermeldung verwendet wird. Auch unter dem Gesichtspunkt der späteren Nutzbarkeit des Frameworks beziehungsweise des Programmiermodelles ist eine konsistente Gestaltung der Schnittstellen wichtig (vgl. [CA06]).

Geprüft werden muss bei dem Versuch, eine Aktion auf dem Modell auszuführen. Dies betrifft das Hinzufügen, Konfigurieren und Löschen von Komponenten und das Anlegen und Löschen von Verbindungen. Führt die Prüfung zu einem negativen Ergebnis, darf die Aktion entweder nicht ausgeführt werden oder muss rückgängig gemacht werden. Dies ist abhängig davon ob vor dem Ausführen einer Aktion geprüft wird, oder das Modell nach einer erfolgten Änderung auf Konsistenz mit den formulierten Beschränkungen geprüft wird. Beide Realisierungsformen sind denkbar, in der prototypisch implementierten Variante wird zunächst die Prüfung durchgeführt und eine Aktion auf dem Modell nur dann ausgeführt, wenn sie zulässig ist. Diese Variante hat den Vorteil, dass keine unnötigen Aktualisierungen der an das Modell angeschlossenen Sichten ausgelöst werden.

Es bleibt festzulegen, auf welche Informationen die Prüfmethoden Zugriff haben müssen, um ihre Aufgaben erfüllen zu können. Da die Informationen zu einer Komponente und zu den Verbindungen innerhalb der Modellrealisierung in den Klassen `ComponentInfo` und `ConnectionInfo` gekapselt sind, benötigen die entsprechenden Prüfmethoden Zugriff auf die Instanzen dieser Klassen. Die als

Argumente übergebenen Instanzen müssen dabei – da ja vor der Ausführung der Aktion auf dem Modell geprüft werden soll – bereits die angestrebten Änderungen des Modellzustandes reflektieren. Darüber hinaus muss ein Zugriff auf den aktuellen Modellzustand möglich sein, da einige Prüfungen nicht nur mit den „lokalen“ Informationen durchgeführt werden können. Hierunter fällt beispielsweise die Frage, ob eine Instanz eines Komponententyps gelöscht werden kann, von dem mindestens eine Instanz im Modell vorhanden sein muss. Da dieser Zugriff auf das Modell mehrfach benötigt wird, wird die Referenz auf das Modell nicht jeweils als Parameter der Prüfmethoden realisiert, sondern als Parameter des Konstruktors.

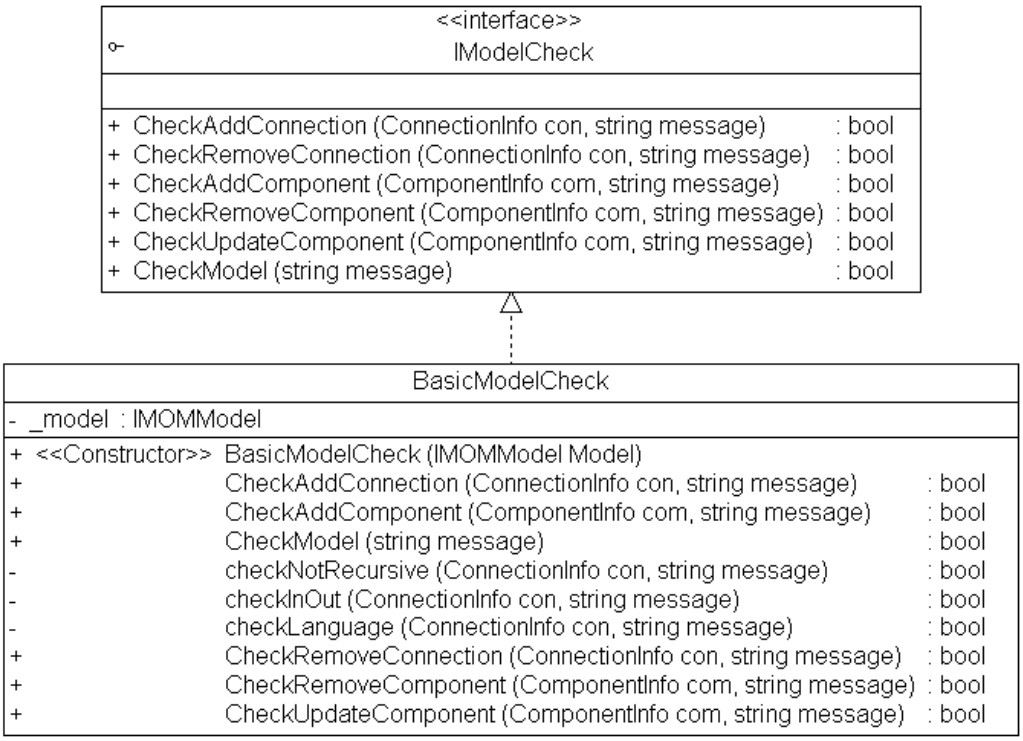


Abb. 8.9: Das Interface IModelCheck und die prototypische Implementierung BasicModelCheck

Die obigen Überlegungen führen zu dem in Abb. 8.9 dargestellten Interface `IModelCheck`. Die Methode `CheckModel()` dient dazu, das gesamte Modell auf Konsistenz zu prüfen. Dies geschieht beispielsweise vor der Generierung einer Anwendung auf Basis des Modells. Im Falle einer Anwendung der hier vorgestellten Ergebnisse im Rahmen von Simulationsanwendungen könnte diese Methode vor dem Start der Berechnungen ausgeführt werden.

Die ebenfalls in Abb. 8.9 gezeigte Klasse `BasicModelCheck` stammt aus der vorgenommenen prototypischen Realisierung und zeigt die Kapselung einzelner Tests in Form privater Methoden, deren Ergebnisse dann, durch logische Verknüpfungen verbunden, das Gesamtergebnis einer Prüfung liefern.

Zusammen mit der Möglichkeit, Sprachbeschreibungen für die zwischen Komponenten austauschbaren Nachrichten anzugeben, lässt sich ein breites Spektrum an Einschränkungen für ein Modell formulieren.

8.5 Sichten

Damit ein Endnutzer ein Softwaresystem auf Basis des in diesem Kapitel entworfenen Programmiermodells verwirklichen kann, muss er in der Lage sein, eine konkrete Ausprägung dieses Modells zu beschreiben. Zu diesem Zweck ist eine geeignete Möglichkeit der Interaktion vonnöten. Im Folgenden werden diese Interaktionsmöglichkeiten als Sichten im Sinne des in Kapitel 4 beschriebenen MVC'-Musters entworfen.

Die Sichten können über die in der Schnittstelle der Modellrealisierung deklarierten Methoden auf das Modell und seine Daten zugreifen. Der Umfang dieser Schnittstelle bestimmt daher auch die Aktionen, mit denen ein Nutzer Änderungen an dem Modell vornehmen kann und muss dementsprechend sorgfältig entworfen werden. Die Entwicklungsumgebung benötigt keine Kenntnisse über diesen Teil der Schnittstelle des Modells, da für die Integration einer Sicht seitens der Entwicklungsumgebung nur die Schnittstellen `IModelObserver` beziehungsweise `IView` verwendet werden (siehe Abschnitt 4.4.2 auf Seite 72). Hier zeigt sich ein Vorteil des in Kapitel 4 entworfenen MVC'-Musters. Die Kommunikation der Sichten mit dem Modell ist für die Entwicklungsumgebung nur soweit von Belang, wie es die Benachrichtigungen im Falle von Änderungen betrifft, da sie die hierfür notwendigen Verknüpfungen anlegen muss.

Die Sichten auf ein Modell können hinsichtlich ihres Zugriffs auf das Modell in drei grundlegende Gruppen unterteilt werden. Dies sind zum einen Sichten, die nur eine Darstellung des Modellzustand realisieren (lesender Zugriff). Daneben gibt es noch Sichten, die sowohl den Modellzustand darstellen als auch seine Manipulation erlauben (lesender und schreibender Zugriff) und als dritte Gruppe Sichten, die nur Änderungen am Modellzustand ermöglichen, aber keine eigenständige Anzeige bieten (schreibender Zugriff). Auch wenn es sich bei der letzten Gruppe nicht um Sichten im ursprünglich diskutierten Sinne handelt, ermöglicht das MVC'-Muster deren Umsetzung. Im Zusammenspiel mit anderen Sichten können diese ausschließlich schreibenden Sichten vorteilhaft eingesetzt werden. Für die prototypische Realisierung des nachrichtenbasierten Programmiermodells wurden Exemplare aller drei Gruppen umgesetzt.

Eine Sicht, welche einen Teilaspekt des Modellzustandes darstellen kann und auch Änderungen erlaubt, wurde mit dem Eigenschaftsfenster in Abb. 8.6 bereits angesprochen. Eine sinnvolle Nutzung dieser Sicht ist jedoch erst mit anderen Sichten möglich, da zunächst eine Komponenteninstanz selektiert werden muss. Dies ist unter alleiniger Verwendung des Eigenschaftseditors nicht möglich.

Die visuelle Repräsentation als Graph stellt ebenfalls den Zustand des Modells dar und erlaubt Änderungen an diesem. Sie ist beim derzeitigen Stand der prototypischen Realisierung die zentrale Sicht. Diese Entwurfsentscheidung beruht auf der Eignung des Paradigmas der visuellen Programmierung für das Einsatzfeld der Endnutzerprogrammierung [Bur99]. Da eine alleinige Realisierung von komplexeren Anwendungen auf Basis des visuellen Paradigmas an Grenzen stößt [Sch98], wird diese Sicht durch andere Darstellungen ergänzt. Diese Möglichkeit ergibt sich direkt aus dem MVC'-Muster, für besondere Aspekte eines Modells oder spezielle Aufgaben können angepasste Sichten bereitgestellt werden, um den Anwender zu unterstützen. Dies ist im Hinblick auf die Verwendung im Rahmen der Endnutzerprogrammierung ein positiver Aspekt. An dieser Stelle sei nochmals darauf hingewiesen, dass eine schrittweise Erweiterung um zusätzliche Sichten möglich ist.

In Abb. 8.10 ist ein Ausschnitt der Graphdarstellung[76] des nachrichtenbasierten Modells in der Cobamos IDE zu sehen. Der hier dargestellte Ausschnitt basiert wiederum auf der in der Einleitung

[76] Die Implementation dieser Sicht verwendet zur Visualisierung des Graphen eine leicht modifizierte Version der Anzeigekomponente und einige andere Komponenten aus der Netron Graphbibliothek, Version Sommer 2005, [Van05].

dieser Arbeit bereits angesprochenen Anwendungsfamilie zum Umgang mit Datenströmen aus ganzen Zahlen. Neben zwei Instanzen der Generatorkomponente zum Erzeugen von Zahlen, ist eine Instanz der Komponente zum Anwenden von Berechnungsvorschriften auf Zahlen („Evaluator“) zu sehen sowie eine Instanz der Komponente zur Anzeige von empfangenen Zahlen („View“). Eine genauere Behandlung dieser Beispielanwendung des nachrichtenbasierten Programmiermodells findet sich in Kapitel 9.

Der dargestellte Graph zeigt die Verbindung mehrerer Komponenteninstanzen. Neben dem Typ der verwendeten Komponente wird der Bezeichner der Instanz angezeigt. Die Verbindungen zeigen den möglichen Nachrichtenfluss zwischen den Komponenteninstanzen an, die Pfeilspitzen verdeutlichen dabei die Richtung des Nachrichtenflusses. Das eingeblendete Fenster „Eigenschaften“ zeigt die durch den Nutzer konfigurierbaren Eigenschaften der gewählten Komponenteninstanz – in diesem Fall vom Typ „Evaluator“ – an. Für diese Instanz ist noch kein Name vergeben worden, daher wird hier im Graphen noch die intern zur Identifikation verwendete GUID angezeigt. Die menschenlesbaren Bezeichner sind – wie bei der Diskussion des Modellentwurfes in Abschnitt 8.2 angesprochen – nur als Erleichterung für den Benutzer gedacht, sie werden nicht zur Identifikation herangezogen.

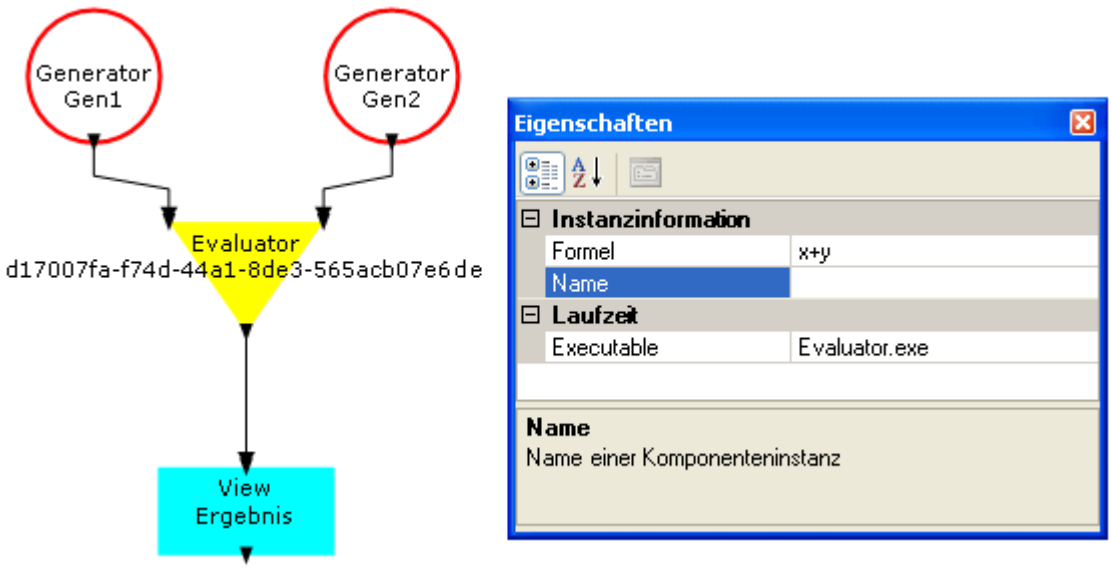


Abb. 8.10: Graphdarstellung und Eigenschaftsfenster in der Cobamos- Entwicklungsumgebung

Die bei Verwendung des in Kapitel 4 entworfenen MVC'-Musters für die Integration der Graphansicht in die Entwicklungsumgebung relevante Anforderung ist die Implementation der Schnittstelle `IView`. Dadurch wird die realisierende Klasse als Sicht erkannt und beim Laden einer Komponente dementsprechend instanziert. Dabei wird die Benachrichtigungsmethode der Graphsicht beim Modell registriert, so dass die Sicht immer den aktuellen Modellzustand darstellen kann. Um das Entwickeln von Sichten zu vereinfachen, wurde die abstrakte Klasse `AbstractView` implementiert. Die Klasse `AbstractView` stellt eine Standardimplementation der Schnittstelle `IView` zur Verfügung und realisiert darüber hinaus die Funktionalitäten, die für die Nutzung eines Formulars als Kindfenster in einer MDI-Umgebung notwendig sind.

Damit in einer Sicht auf Änderungen am Modell reagiert werden kann, muss die geerbte Methode `OnModelChanged()` – dies ist für die Graphsicht in Codebeispiel 8.3 gezeigt – überschrieben werden.

Wenn dieses Ereignis ausgelöst wird, wurde eine Änderung am Zustand des Modells durchgeführt, die unter Umständen eine Aktualisierung der Sicht erfordert.

```
(1)  public override void OnModelChanged(object sender,
                                Cobamos.Ide.Model.UserModelChangedEventArgs args)
(2)  {
(3)     base.OnModelChanged(sender, args);
(4)     this.synchronizeGraph();
(5)  }
```

Codebeispiel 8.3: Die Methode OnModelChanged der Graphsicht

Die eigentliche Funktionalität für die Synchronisation des Graphen mit dem Modell ist in der Methode `synchronizeGraph()` gekapselt. Da deren Quelltext sehr umfangreich ist und nicht ohne eine weitergehende Erläuterung der zur Visualisierung verwendeten Graphbibliotheken auskommt, werden nur der hierfür verwendete Algorithmus und die zu beachtenden Punkte angesprochen.

```
(1)  Für jede Komponenteninstanz im Modell
(2)     Gibt es eine Darstellung?
(3)        Nein: anlegen
(4)     Aktualisiere angezeigte Eigenschaften im Graphen
(5)  Für jede Verbindung im Modell
(6)     Gibt es eine Darstellung?
(7)        Nein: anlegen
(8)  Für jede dargestellte Komponenteninstanz
(9)     Existiert eine Entsprechung im Modell?
(10)       Nein: Löschen der Darstellung
(11) Für jede dargestellte Verbindung
(12)    Existiert eine Entsprechung im Modell?
(13)       Nein: Löschen der Darstellung
```

Codebeispiel 8.4: Algorithmus zur Synchronisation von Graph und Modell in Pseudocode

Codebeispiel 8.4 zeigt den Algorithmus in Pseudocode. Zunächst wird auf hinzugefügte Elemente im Modell geprüft, die noch nicht im Graphen angezeigt werden. Bei diesem Durchlauf werden auch die im Graphen angezeigten Eigenschaften der Instanzen – dies ist in der prototypischen Realisierung der menschenlesbare Bezeichner – aktualisiert. Anschließend muss auf Elemente geprüft werden, die im Modell eventuell gelöscht wurden, im Graphen aber noch angezeigt werden. Diese naive Herangehensweise stellte sich zunächst als für die prototypische Implementation ausreichend dar. Im Verlauf der Anwendung des hier implementierten Programmiermodells auf konkrete Anwendungsfamilien zeigte sich jedoch, dass dieses Vorgehen bei umfangreichen Anwendungsmodellen nicht effizient genug ist. Eine mögliche Lösung hierfür besteht darin, dass das Modell zumindest die Art der letzten Änderung anzeigt oder diese vollständig – also inklusive der Parameter – an die Sichten weiterreicht. Dadurch müssen die Sichten bei einer Aktualisierung keinen vollständigen Abgleich vornehmen. Um die Art der Änderung anzuzeigen, ist das Einführen von Zustandsanzeigern ausreichend, die auf eine der fünf möglichen Aktionen hinweisen (Hinzufügen und Löschen von Instanzen und Verbindungen sowie Ändern der Instanzeigenschaften). In der Folge kann im obigen Algorithmus einer der vier zu prüfenden Fälle ausgewählt werden; der Aufwand für eine Synchronisation reduziert sich dementsprechend. Damit ein Weitergeben der vollständigen Anweisung, die zur letzten Änderung führte, möglich wird, müsste eine entsprechende abstrakte Befehlssyntax eingeführt werden, die von jeder Sicht auf

ihren eigenen Datenhintergrund angewandt werden kann. Diese Lösungsansätze wurden jedoch noch nicht weiter verfolgt.

Eine weitere realisierte Sicht verwendet eine formularbasierte Darstellung. Diese Sicht bietet zum derzeitigen Entwicklungszeitpunkt nur eine Möglichkeit den Modellzustand zu ändern und verwendet zu dessen Visualisierung eine Übersichtsdarstellung der Graphsicht. Um die Eingaben für einen Anwender zu vereinfachen, ist eine Autovervollständigung integriert worden, welche Vorschläge für Befehle und mögliche Parameter bereitstellen kann. In Abb. 8.11 ist die Formularansicht zu sehen. Die dargestellte Sitzung startete mit einem leeren Modellzustand, alle Änderungen wurden über das Formular vorgenommen. Hier ist auch zu sehen, dass die Graphsicht den über das Formular erzeugten Zustand des Modells reflektiert, auch die vergebenen Bezeichner werden in dieser Sicht dargestellt. In der aktuell bearbeiteten letzten Zeile des Formulars ist das Anlegen einer Verbindung zu sehen. Die Autovervollständigung schlägt die beiden im Modell vorhandenen Instanzen vor, deren Bezeichner mit den eingegebenen Buchstaben beginnen. Es ist jeweils nur die aktuelle Zeile editierbar, beim Verlassen der Zeile wird die jeweilige Aktion ausgeführt und die Zeile in einen reinen Anzeigemodus umgeschaltet. Das Formular erlaubt eine interaktive Manipulation des Modells mit einer textbasierten Notation. Die formularbasierte Darstellung wurde aufgrund der in der Literatur geschilderten positiven Erfahrungen mit dieser Darstellungsform im Einsatzfeld der Endnutzerprogrammierung (vgl. Abschnitt 2.5.1) gewählt.

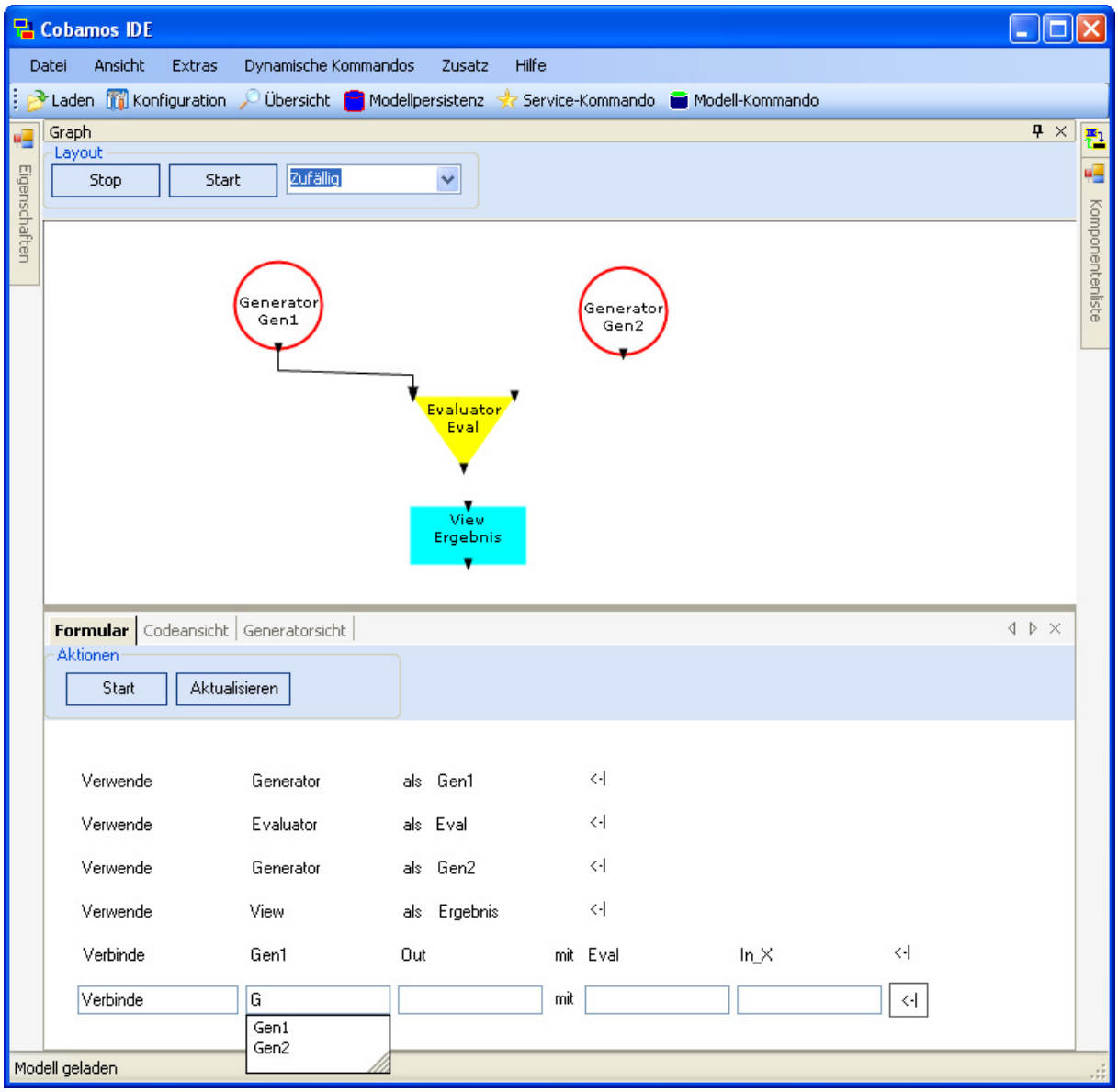


Abb. 8.11: Formularbasierte Sicht und Graphansicht

Auch aus der noch nicht behandelten dritten Gruppe von Sichten, den reinen Repräsentationen des Modellzustandes, sind Beispiele implementiert worden. Vorgestellt werden soll hier eine Sicht zur textuellen Darstellung des Modellzustandes, da sich hieran auch die Realisierung unterschiedlicher Transformationen erläutern lässt.

In Kapitel 4 wurde bereits die mögliche Notwendigkeit von Transformationen des Modells zur Darstellung des Zustandes durch die Sichten angesprochen. Falls eine Sicht nur das Ergebnis einer einzelnen Transformation anzeigen kann und diese Transformation auch nur für diese Sicht benötigt wird, spricht nichts dagegen, dass eine enge Kopplung der Transformation mit der Sicht besteht. Je nach Umfang der für die Transformation benötigten Realisierung kann diese dann sogar in die Realisierung der Sicht integriert werden.

Im folgenden Abschnitt soll ein Ansatz vorgestellt werden, der dazu dient, eine Sicht mit unterschiedlichen Transformationen konfigurieren zu können. Diese Problemstellung trat bei der Realisierung einer Sicht mit einer textbasierten Darstellung des Modellzustandes auf. Die zur Präsentation einer textuellen Darstellung benötigte Funktionalität ist im Wesentlichen unabhängig von ihrem Inhalt. Aus

diesem Grund ist es sinnvoll, diese wiederzuverwenden, falls mehrere Sichten dieser Art realisiert werden sollen. Es wäre möglich, eine Basisklasse mit dem benötigten Funktionsumfang zu erstellen, und die spezialisierten Sichten von dieser abzuleiten; jede der Sichten würde dann eine andere Transformation zum Erzeugen der Darstellung verwenden. Diese Herangehensweise hätte jedoch den Nachteil, dass die eigentlich zur Präsentation benötigte Funktionalität mehrfach geladen werden müsste. Außerdem verwischt diese Kopplung die eigentlichen Zuständigkeiten; die Transformation eines Modellzustandes in ein textbasiertes Format ist unabhängig von der Darstellung der Textdaten.

Um eine Transformation allgemein zu beschreiben und sie durch eine lose Kopplung mit der verwendenden Sicht austauschbar zu gestalten, wird ein Interface entworfen. Die Abbildung eines Modellzustandes auf Textdaten kann direkt als Methode mit entsprechendem Parameter und Rückgabewert ausgedrückt werden. Die Abbildung kann zustandslos sein, jeder erneute Aufruf erzeugt unabhängig vom vorherigen die Textdaten.

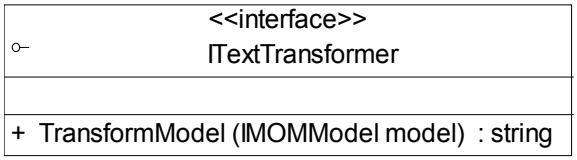


Abb. 8.12: Die Schnittstelle ITextTransformer

In Abb. 8.12 ist die Schnittstelle `ITextTransformer` zu sehen, welche für die Beschreibungen der Modelltransformationen Verwendung findet. Eine allgemeinere Schnittstelle ließe sich durch die Verwendung des Typs `Object` für den Rückgabewert der Methode realisieren. Dies ist in der prototypischen Realisierung umgesetzt worden, findet aber noch keine Verwendung.

Das Codebeispiel 8.5 zeigt die Anwendung der Transformationsmethode. Die Methode `TextRefresh()` wird unter anderem auch bei einer Benachrichtigung über eine Änderung des Modellzustandes aufgerufen. Die Methode wendet die aktuell in den Optionen referenzierte Transformation auf das Modell an und trägt das Ergebnis in das Anzeigefeld ein.

```
(1)  private void TextRefresh(IModel model)
(2)  {
(3)     this.CodeControl.Text = _options.Transform.TransformModel(model as
                                                    Cobamos.MOM.Model.MessageModel);
(4)     this.CodeControl.Refresh();
(5)  }
```

Codebeispiel 8.5: Anwendung der Transformation

Als Transformationen zur Verwendung mit der Textsicht sind eine statistische Auswertung und eine XML-Darstellung des Modellzustandes implementiert. Die Transformation kann während der Nutzung der Anwendung geändert werden, die Darstellung wird dann entsprechend neu erzeugt. Das Erzeugen der textuellen Repräsentationen in den Transformationsmethoden wird durch die Wahl der Datenstruktur für das Modell vereinfacht (siehe Abschnitt 8.2). Sowohl das XML-Format als auch die statistische Auswertung verwenden in der Darstellung getrennte Bereiche für die Informationen zu den Komponenteninstanzen und zu den Verbindungen. Durch die linearen Datenstrukturen im Modell sind diese Daten in einem Durchlauf zu ermitteln. Aber auch textuelle Formate, bei denen die zugehörigen Verbindungen bei einer Komponenteninstanz angegeben werden, können realisiert werden. Aufgrund

des zusätzlichen Indexes im Modell, welcher zu einer Komponenteninstanz alle Verbindungen angibt, an denen sie beteiligt ist, sind auch derartige Anfragen sehr leicht zu beantworten.

8.6 Der Connector

Zur Integration der Sichten und des Modells in die Cobamos-Entwicklungsumgebung wird, wie in Kapitel 4 beschrieben, ein Connector implementiert. Dieser vermittelt insbesondere auch den Zugriff auf die Dienste des Frameworks. Modell und Sichten greifen nicht direkt auf die Dienste zu, die Zugriffe werden über den Connector delegiert. Damit ergeben sich schon zwei grundsätzliche Anforderungen an die Realisierung des Connectors für dieses Programmiermodell. Zum einen muss er – damit eine Nutzung in der Entwicklungsumgebung überhaupt möglich ist – das Interface `IConnector` implementieren. Zum anderen müssen die von den realisierten Sichten und dem Modell benötigten Dienste ermittelt werden und die notwendigen Methoden zum Delegieren der Aufrufe umgesetzt werden.

Es ist nicht für alle Dienste nötig, dass Modell und Sichten Zugriff auf die gesamte Funktionalität erhalten. Dies ist beispielsweise für den Dienst zur Verwaltung von Konfigurationsdaten der Fall. Die eigentliche Nutzung des Dienstes erfolgt durch den Connector, die Bestandteile des MVC'-Musters erhalten nur Zugriff auf die für sie jeweils notwendigen Konfigurationsinformationen. Neben dem Zugriff auf die Konfigurationsdaten wurden in der vorgenommen Realisierung auch Zugriffe auf den Dienst zur Anzeige von Meldungsfenstern, den Dienst zur Ausgabe von Statusmeldungen in der Statuszeile des Hauptfensters und einen Dienst zur Ausgabe von Meldungen in einem besonderen Textfenster der Entwicklungsumgebung implementiert. Der überwiegende Teil dieser Dienste wurde eingebunden, um eine möglichst nahtlose Integration in die Nutzungsoberfläche und die Interaktionskonzepte der Entwicklungsumgebung zu ermöglichen.

Der Vorteil dieses Delegierens von Aufrufen ist wie in Kapitel 4 angesprochen, dass Änderungen an den Zugriffstrukturen der Dienste an einer einzigen Stelle abgefangen werden können und nicht zu Änderungen bei jedem Nutzer führen.

Der Connector übernimmt hier noch eine zusätzliche Aufgabe. Um die Eigenschaften einer in der Graphsicht selektierten Komponenteninstanz im Eigenschaftseditor anzeigen zu können, muss eine Referenz zwischen Graphsicht und Eigenschaftseditor bestehen. Um keine enge Bindung zwischen zwei konkreten Sichten zu erzeugen, wird diese Verbindung über den Connector realisiert. Dieser besitzt Referenzen auf alle Sichten, welche vernetzt sein müssen, und delegiert eventuelle Aufrufe der Sichten untereinander entsprechend weiter. Die Sichten besitzen, da sie die Schnittstelle `IView` implementieren müssen, eine Eigenschaft für eine Referenz auf den Connector. Diese Eigenschaft wird in den zu vernetzenden Sichten derart überschrieben, dass bei ihrer Zuweisung eine Referenz auf die aktuelle Sicht beim Connector eingetragen wird. Dies realisiert die Vernetzung (vgl. zu diesem Punkt auch Fußnote 48 in Abschnitt 4.4.2).

Das Modell, der Connector und die gesamten bisher angesprochenen Sichten sind in einer einzigen Komponente gekapselt und können daher in einem einzigen Aufruf geladen und instanziert werden. Dies ist insbesondere beim Wechsel zwischen verschiedenen Programmiermodellen von Vorteil. Dieser Wechsel kann ohne einen Neustart der IDE erfolgen.

8.7 Anwendungsgenerierung

In den vorangegangenen Abschnitten wurden schrittweise das Modell, einige Sichten und der Connector und damit alle notwendigen Bestandteile des MVC'-Musters entworfen. Dies geschah mit dem Ziel der Umsetzung des in Kapitel 7 konzeptionell vorgestellten nachrichtenbasierten Programmiermodells. Nun müssen die Möglichkeiten zur Generierung einer Softwareapplikation aus einem Modellzustand dieses Programmiermodells erarbeitet werden. Dieses Ziel stand am Anfang der Überlegungen zu dieser Arbeit. Ein Endnutzer ohne tiefgreifende Kenntnisse der Softwareentwicklung soll in die Lage versetzt werden – in einem gewissen Rahmen – Software zu entwickeln oder eine Anwendung über ein reines Konfigurieren hinaus anzupassen.

Die zentrale Idee des betrachteten Programmiermodells ist die Verknüpfung von Komponenteninstanzen durch den Austausch von Nachrichten. Damit dies geschehen kann, ist eine Möglichkeit zur Übertragung von Nachrichten zwischen einzelnen Komponenteninstanzen notwendig. Die Entwurfsidee einer Anwendung, welche dieses leistet, wurde in Abschnitt 7.4.2 beschrieben. Der vorgestellte Ansatz ermöglicht es auch, die in Kapitel 7 vorgestellten Kommunikationsmuster zu realisieren. Zur Realisierung dieses Entwurfes wurde eine als eigenständige Anwendung ausführbare Komponente, der Cobamos Nachrichtenvermittler, entworfen. Diese Anwendung verwaltet und verknüpft Nachrichtenwarteschlangen und stellt damit die Infrastruktur für den Nachrichtenaustausch zwischen deren Komponenteninstanzen bereit. Der nachfolgende Abschnitt geht auf einige Details der Realisierung ein.

8.7.1 Realisierung des Nachrichtenvermittlers

Der Cobamos Nachrichtenvermittler bildet die Grundlage für die Kommunikation der Komponenteninstanzen, aus denen sich die erzeugte Endanwendung[77] zusammensetzt. Dies bedeutet, dass er als Bestandteil in jeder Anwendung vorkommt, die auf Basis des in diesem Kapitels beschriebenen Programmiermodells realisiert wird. Um eine Endanwendung nicht zu eng an eine konkrete Realisierung dieses Mechanismus zur Nachrichtenübermittlung zu binden, soll der Aufruf von Funktionalität des Nachrichtenvermittlers – wie in Abschnitt 7.4.2 dargelegt – über Nachrichten erfolgen. Dies geschieht mit dem Ziel, eine lose Kopplung zwischen den Anwendungsteilen zu erreichen. Damit diese Idee realisiert werden kann, muss zunächst festgelegt werden, welche Funktionalität des Vermittlers über Nachrichten ansprechbar sein soll. Anschließend muss eine geeignete Nachrichtensprache zur Kommunikation zwischen Komponenten oder Anwendungen und dem Vermittler entworfen werden.

Zunächst muss es möglich sein, dem Nachrichtenvermittler mitzuteilen, welche Warteschlangen er hinsichtlich des Eintreffens von Nachrichten überwachen soll. Daraus ergibt sich direkt, dass auch das Abmelden von solchen Eingangswarteschlangen (aus Sicht des Vermittlers) notwendig ist. Damit die Nachrichten an andere Komponenteninstanzen weitergegeben werden können, müssen sich diese als Interessenten für eine Nachrichtenquelle an- und abmelden können. Die für diese Aufgaben entwickelte Nachrichtensyntax basiert auf XML. Um eine einfache Nutzung bei der Entwicklung von Komponenten zur Verwendung mit dem Cobamos Nachrichtenvermittler zu ermöglichen, wurde eine Bibliothek realisiert, welche eine Reihe von Methoden zum Generieren der benötigten Nachrichten zur Verfügung stellt. Eine Anpassung der Nachrichtensyntax kann für alle Komponenten, welche die Bibliothek verwenden, durch einen Eingriff in den Quelltext der Bibliothek ausgeführt werden. Die Schnittstelle ergibt sich direkt aus den eben angesprochenen Anforderungen, die Methoden erhalten

[77] Der Begriff Endanwendung wird hier im Sinne der Anwendung, die ein Endnutzer mit dem Werkzeug beschreibt, verwendet. Dies entspricht der untersten Ebene in der Übersicht über den Gesamtkontext der Ergebnisse dieser Dissertation, wie sie in Abb. 1.2 auf Seite 12 in Kapitel 1 dargestellt ist.

jeweils als Parameter die Warteschlangen. Als Rückgabetyp für die Methoden wird die Klasse `System.Messaging.Message` verwendet, diese kapselt eine Nachricht.

MSMQUtils	
+ GetUniqueQueue (string pathToQueue)	: System.Messaging.MessageQueue
+ GetQueue (string pathToQueue)	: System.Messaging.MessageQueue
+ GetExistingQueue (string pathToQueue)	: System.Messaging.MessageQueue
+ CreateRegisterIncomingMessage (string pathToIncoming)	: System.Messaging.Message
+ CreateRegisterReceiverMessage (string pathToIncoming, string pathToReceiver)	: System.Messaging.Message
+ CreateUnregisterReceiverMessage (string pathToIncoming, string pathToReceiver)	: System.Messaging.Message
+ CreateUnregisterIncomingMessage (string pathToIncoming)	: System.Messaging.Message

Abb. 8.13: Die Klasse MSMQUtils

Neben den Methoden zum Erzeugen der Nachrichten stellt die in Abb. 8.13 gezeigte Klasse `MSMQUtils` auch einige Methoden bereit, die die zur Anforderung einer Nachrichtenwarteschlange nötige Funktionalität beinhalten.

Die gesamte Implementation basiert auf der Verwendung von Warteschlangen, derzeit wird dafür das Produkt MSMQ der Firma Microsoft [Mic06j] verwendet. Dies ist allerdings keine Einschränkung, da keine hersteller- oder produktspezifischen Eigenschaften beim Entwurf Verwendung finden. Eine Anpassung auf eine alternative Technologie zur Realisierung der eigentlichen Warteschlangen ist daher möglich.

Der Nachrichtenvermittler erlaubt sowohl, dass sich mehrere Interessenten für eine Quellwarteschlange anmelden, als auch die Anmeldung eines Interessenten bei mehreren Nachrichtenwarteschlangen. Dadurch können die in Kapitel 7 angesprochenen Kommunikationsmuster umgesetzt werden. Damit dies realisiert werden kann, muss der Nachrichtenvermittler zu einer Eingangswarteschlange eine Liste der als Interessenten angemeldeten Warteschlangen speichern können. Als Basisdatenstruktur wird eine Hashtabelle verwendet, diese ist mit den Pfaden der als Nachrichtenquellen registrierten Warteschlangen indiziert. Die Werteeinträge sind jeweils die Listen mit den registrierten Interessentenwarteschlangen. Trifft eine Nachricht in einer der Quellwarteschlangen ein, so wird sie an alle Warteschlangen in der entsprechenden Liste weiterverteilt.

Das bewirkte Verhalten, dass ein Interessent sich für bestimmte Nachrichtenquellen registrieren kann und dann benachrichtigt wird, ist mit dem durch das Observer-Muster (vgl. Kapitel 4) realisierten Verhalten vergleichbar. Allerdings ist der Grad der Unabhängigkeit zwischen Quelle und Beobachter in diesem Szenario höher als im Falle der vorgestellten objektorientierten Realisierung, da die Quelle für die Benachrichtigungen keinerlei Referenz auf den Observer benötigt. Der Observer muss ebenfalls keine Kenntnis der Instanz haben, welche die Benachrichtigungen auslöst bzw. die Nachrichten versendet.

Da der Cobamos Nachrichtenvermittler eine Hintergrundaufgabe wahrnimmt, wurde auch die Nutzungsoberfläche der Anwendung in einer entsprechenden Form realisiert. Sie besitzt kein eigenes Fenster, sondern wird während der normalen Ausführung nur als Symbol in der Benachrichtigungsfläche der so genannten Taskbar angezeigt. In Abb. 8.14 ist das Symbol (am linken Rand der Benachrichtigungsfläche) sowie das aktivierte Menü des Nachrichtenvermittlers zu sehen.

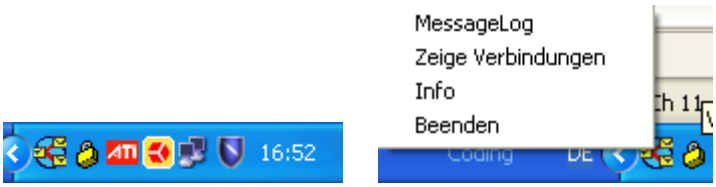


Abb. 8.14: Der Cobamos Nachrichtenvermittler während seiner Ausführung

Die Anforderung, auch die Realisierung von Filtern und vergleichbarer Funktionalität zu erlauben, erfüllt der bisher vorgestellte Entwurf des Nachrichtenvermittlers noch nicht. Da im Vermittler alle ausgetauschten Nachrichten in einer auf dieser Basis realisierten Anwendung weiterverteilt werden, ist es sinnvoll, die Verantwortung für die Filterfunktionalität auch an dieser Stelle anzusiedeln.

Um die Filter austauschbar zu machen, wird das bereits an mehreren Stellen in dieser Arbeit erfolgreich eingesetzte Komponentenkonzept verwendet. Analog zum Vorgehen beispielsweise bei den dynamisch ladbaren Kommandokomponenten wird auch hier ein Interface festgelegt, welches von einem Filter zu implementieren ist. Da ein Nachrichtenfilter sich durch eine Methode beschreiben lässt, welche eine Nachricht als Parameter erhält und eine ebensolche als Rückgabewert liefert, deklariert die Schnittstelle nur eine solche Methode.

Beim Start des Cobamos Nachrichtenvermittlers durchsucht dieser ein vorgegebenes Verzeichnis und überprüft die dort gefundenen Assemblys, ob sie Implementationen des Interfaces enthalten. Ist dies der Fall, so werden die entsprechenden Klassen instanziert. Der Nachrichtenvermittler bietet die Möglichkeit, Filter für einzelne Eingangswarteschlangen zu definieren, diese fand jedoch in den bisherigen Anwendungen keine Verwendung. Es ist ebenfalls möglich einen globalen Filter zu definieren. Hierbei können mehrere Filtermethoden hintereinander geschaltet werden, so dass eine Kette von Filtern entsteht.

Eine prototypische Umsetzung einer Typisierung der Nachrichtenkanäle einer Anwendung durch einen entsprechenden globalen Filter im Nachrichtenvermittler wurde umgesetzt. Auch diverse Filter, welche die bearbeitete Nachricht nicht verändern sondern als Eingabe für eine andere Aufgabe verwenden – zum Beispiel eine Protokollierung – wurden implementiert. Die Gesamtheit der Möglichkeiten, die sich durch den Einsatz von Filtermethoden für die Nachrichten eröffnen, und die Implikationen für das nachrichtenbasierte Programmiermodell konnten leider im Rahmen der vorliegenden Arbeit nicht weiter untersucht werden. Dieser Bereich scheint aber – wie auch schon bei der Diskussion des nachrichtenbasierten Programmiermodells angesprochen – einige interessante Anwendungen zu erlauben.

Der noch fehlende Teilaspekt auf dem Weg zu einer ausführbaren Anwendung ist die Abbildung des Modells auf eine Struktur aus Nachrichtenwarteschlangen und Komponenteninstanzen. Dies wird im folgenden Abschnitt angesprochen.

8.7.2 Generator für die Startanwendung

Im vorangegangenen Abschnitt wurde mit dem Entwurf des Nachrichtenvermittlers die Grundlage für den Nachrichtenaustausch zwischen den Komponenten vorgestellt. Für die Umsetzung eines Modellzustandes in eine ausführbare komponentenbasierte Anwendung müssen jedoch auch die benötigten Komponenteninstanzen und alle notwendigen Warteschlangen erzeugt werden. Zudem müssen die für die Kommunikation der Instanzen nötigen Verknüpfungen der Warteschlangen durch den Nachrichtenvermittler vorgenommen werden. Auch die im Modell festgelegten Eigenschaften der Komponenteninstanzen (vgl. Abschnitt 8.3) müssen entsprechend übertragen werden.

Es ist ein denkbarer Ansatz, dass die Entwicklungsumgebung diese Aufgaben übernimmt. Die notwendige Funktionalität um eine Anwendung auszuführen könnte dann beispielsweise durch ein dynamisch ladbares Kommando mit Zugriff auf das Modell – ähnlich dem Persistenzkommando in Abschnitt 6.6 – realisiert werden. Dieser Ansatz hat jedoch den Nachteil, dass die Entwicklungsumgebung ausgeführt und ein Modellzustand geladen werden muss, um eine Anwendung auszuführen. Auch wenn dies in einigen Situationen – beispielsweise um während der Entwicklung einer Applikation diese zu testen – ein wünschenswertes Verhalten ist, so soll hier dennoch eine Möglichkeit entwickelt werden, die modellierten Anwendungen unabhängig von der Entwicklungsumgebung auszuführen.

Zu diesem Zweck wird eine eigenständige Startanwendung generiert, welche die eingangs angesprochenen Aufgaben übernimmt und unabhängig von der Entwicklungsumgebung und dem Cobamos-Framework ausgeführt werden kann. Damit der hierfür benötigte Generator im Cobamos-Framework in möglichst vielen Anwendungsszenarien zum Einsatz kommen kann, ist dieser als eigenständige Bibliothek realisiert worden. Im Anschluss an die Vorstellung des Generatorkonzeptes wird der verwendete Ansatz zur Integration des Generators in die Entwicklungsumgebung vorgestellt.

Etwas vereinfacht ist es die Aufgabe des Generators, aus Eingangsdaten – im hier betrachteten Szenario ist dies der Modellzustand, also die Beschreibung einer Softwareapplikation – in einem Zwischenschritt übersetzbaren Programmcode zu erzeugen und diesen dann auch zu übersetzen[78]. Der letzte Schritt, der Vorgang der Compilierung, ist für unterschiedliche Eingangsformate im Wesentlichen identisch. In jedem Fall wird dem Compiler Quellcode in einer von ihm unterstützten Programmiersprache zur Übersetzung übergeben. Konfigurierbar muss also insbesondere der erste Schritt dieses Prozesses, die Erzeugung des Programmcodes aus den Eingangsdaten, sein.

Dieser erste Prozessschritt kann als Abbildung von Eingangsdaten auf Programmcode in einer unterstützten Programmiersprache gesehen werden. Der Compiler benötigt für den zweiten Schritt den Quelltext und unter Umständen noch eine Reihe von Informationen (beispielsweise zu referenzierende Bibliotheken) und Konfigurationsdaten. Die unten stehende Abb. 8.15 verdeutlicht diese Sichtweise des Generierungsprozesses als zweistufigen Prozess. Die beiden unterschiedlichen Eingangstransformationen zeigen dabei den konfigurierbaren Teil des Gesamtprozesses an.

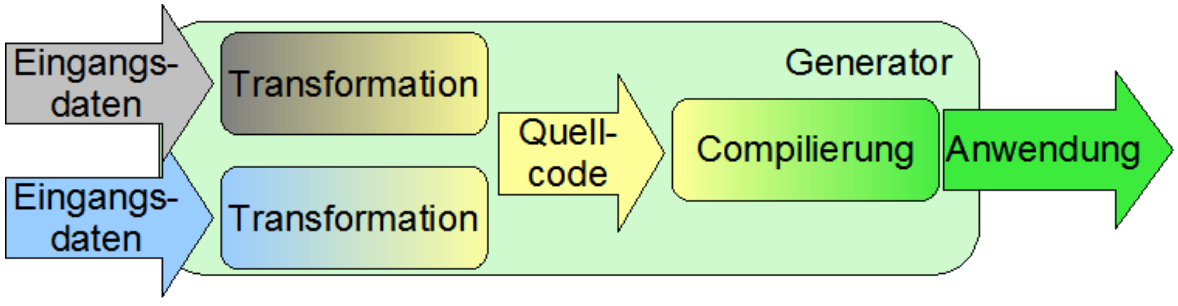


Abb. 8.15: Generierung der Anwendung als zweistufiger Prozess

Um die eigentliche Abbildung der Eingangsdaten des Generators auf den zu compilierenden Quellcode austauschbar zu gestalten, wird ein entsprechendes Interface definiert. Dessen zentrale Methode ist die Methode `GetCodeString()`, welche einen Parameter vom Typ `Object` erhält und einen `String` als

[78] Eine Einführung in das Thema Generatoren, sowie einen Überblick über vorhandene Ansätze und die Vor- und Nachteile dieser Technik ist in Appendix A zu finden. Auch die für die Realisierung des Generators verwendete Technik auf Basis einer Kombination aus XML und XSLT wird in diesem Anhang motiviert.

Rückgabewert liefert. Der Parameter repräsentiert die Eingabedaten, der Rückgabewert enthält das Ergebnis der Transformation als Zeichenkette. Neben dieser Methode existieren weitere Methoden zur Festlegung und Abfrage von Compilereinstellungen.

Zur Zeit unterstützt die prototypische Realisierung des Generators die Zielsprachen C# und VB.NET als Eingangssprachen für den Übersetzungsschritt. Prinzipiell sind jedoch alle anderen .NET-Sprachen möglich, für die ein entsprechender CLR-kompatibler Compiler zur Verfügung steht. Innerhalb des .NET-Frameworks existieren im Namensraum `System.CodeDom.Compiler` Abstraktionen der Compiler, welche sich aus Programmen heraus nutzen lassen. Die Klassen dieses Namensraums werden auch zur Realisierung des hier beschriebenen Generators verwendet. Da das Interface `ICodeCompiler` als gemeinsame Schnittstelle für die Compilerklassen einen uniformen Zugriff auf deren Funktionalität ermöglicht, können auch eventuell später ergänzte Compiler analog zu den bereits integrierten verwendet werden.

Der Generator verwendet eine Realisierung der Schnittstelle `ICodeAdapter` um die beim Aufruf übergebenen Daten zu transformieren. Die Codeadapter sind als Komponenten realisiert, durch verschiedene Realisierungen ist hier ein breites Spektrum an Eingabeformaten für den Generator denkbar. So könnte ein Adapter implementiert werden, der mittels der Eingangsdaten den Programmcode aus einer Datenbank abfragt oder diesen über einen Webservice ermittelt. Realisiert sind zum bisherigen Zeitpunkt Adapter für die direkte Compilierung von C#-Quellcode in Form von Dateien oder Zeichenketten und zur Transformation von XML-basierten Eingaben. Der XML-Adapter verwendet ein XSLT-Stylesheet zur Transformation der Eingabedaten und ist daher an unterschiedliche XML-Formate anpassbar.

Da bereits eine Sicht existiert, welche den gesamten Modellzustand in einem XML-basierten Anzeigeformat darstellt, liegt es nahe, diese Daten in Kombination mit dem XML-Codeadapter zur Erzeugung der Startanwendung zu verwenden.

Die Abbildung des Modellzustandes auf eine Softwareapplikation kann im Wesentlichen analog zur Abbildung auf ein Darstellungsformat gesehen werden, daher erfolgt die Integration des Generators in die Entwicklungsumgebung durch die Implementation einer entsprechenden Sicht. Anders als die bisher vorgestellten Sichten erlaubt diese weder eine Manipulation des Modellzustandes, noch bietet sie eine visuelle Repräsentation für diesen. Da das Generieren der Anwendung ein vergleichsweise aufwendiger Vorgang ist, wird diese nicht bei jeder Änderung des Modellzustandes neu erzeugt.

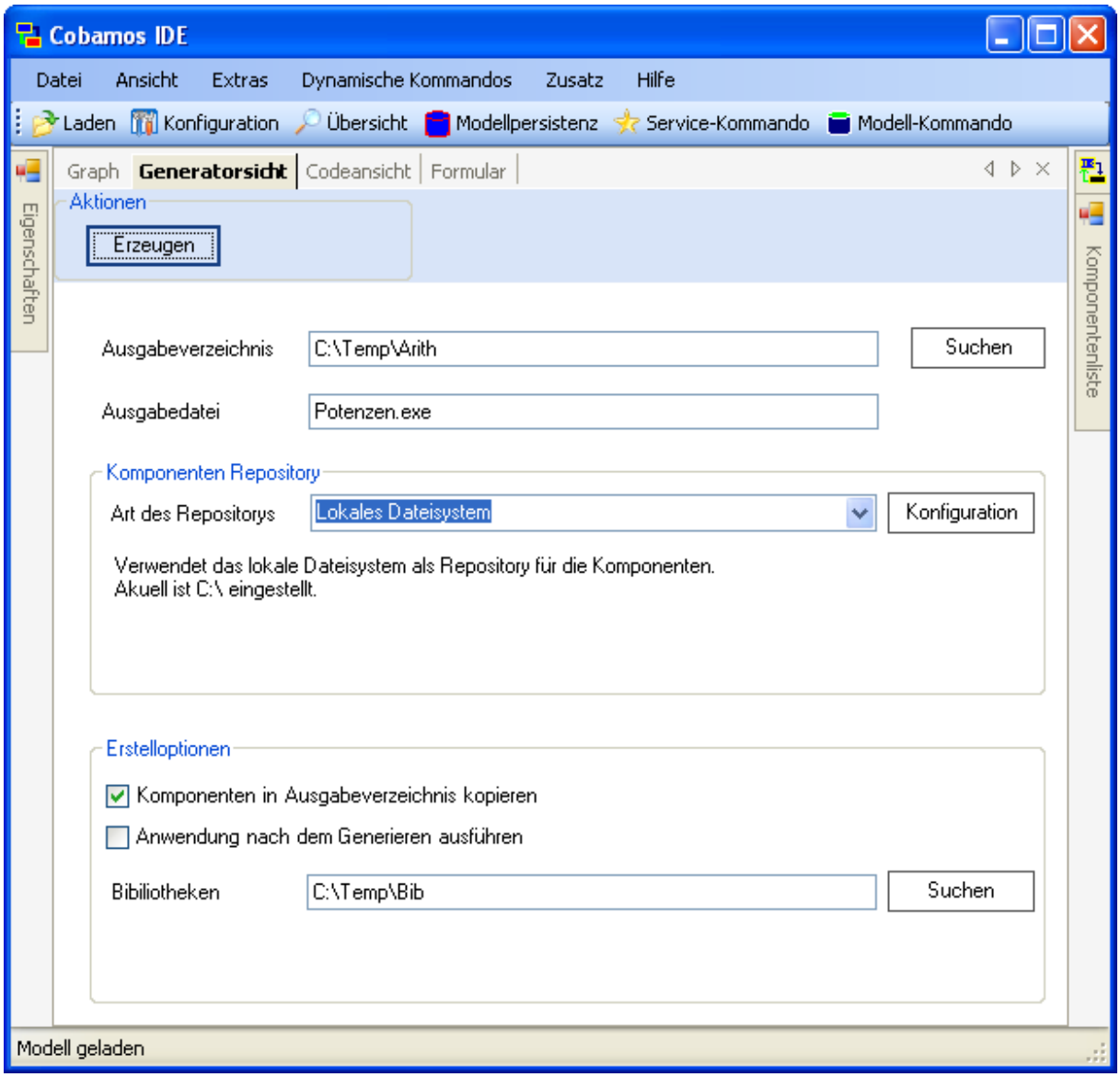


Abb. 8.16: Die Generatorsicht

Diese in Abb. 8.16 dargestellte Generatorsicht ermöglicht es, einige Konfigurationseinstellungen für den Generierungsprozess festzulegen. Da die Startanwendung die Komponenten instanzieren muss, ist es notwendig, dass ein Zugriff auf die Komponentenrealisierung möglich ist. Um hier unterschiedliche Anwendungsmöglichkeiten – wie beispielsweise die bereits angesprochene zentrale Bereitstellung der Komponenten – zuzulassen, können verschiedene so genannte Repositorys mit dem Generator verwendet werden. Dabei handelt es sich um Speicher für Komponenten, die über eine Schnittstelle angesprochen werden können.

Da die generierte Startanwendung die Struktur für den Nachrichtenaustausch in der Anwendung unabhängig von den konkreten Komponentenrealisierungen festlegt, können diese Realisierungen unter gewissen Voraussetzungen ausgetauscht werden, ohne dass ein erneutes Erzeugen der Startanwendung notwendig wird. Damit dies möglich ist, muss sich die neue Version in der selben Weise wie die vorherige instanzieren lassen und über identische Ports verfügen. Diese Bedingungen können inkompatibles Verhalten einer neuen Komponentenversion nicht ausschließen, dies ist entsprechend zu beachten.

8.8 Modellierbare Systeme

Nachdem in diesem Kapitel eine Umsetzung des in Kapitel 7 beschriebenen nachrichtenbasierten Programmiermodells vorgestellt wurde, soll in diesem Abschnitt kurz auf die Systeme eingegangen werden, welche sich auf dieser Basis realisieren lassen. Diese Diskussion erhebt dabei keinen Anspruch auf Vollständigkeit, aufgrund der Vielzahl von denkbaren Gestaltungsmöglichkeiten für Softwaresysteme ist dies auch kaum zu leisten. Neben der konzeptionellen Betrachtung in diesem Abschnitt wird im folgenden Kapitel 9 eine Anwendung der bisherigen Ergebnisse auf konkrete Anwendungsszenarien vorgestellt.

Das hier betrachtete Programmiermodell kann abstrahiert als Graph gesehen werden, in dem Komponenteninstanzen als Knoten und Nachrichtenkanäle als Transitionen betrachtet werden. Die Nachrichten in diesem Graphen fließen von Knoten zu Knoten und es gibt keine Einschränkungen seitens des Modells hinsichtlich der Anzahl, der zu einem Zeitpunkt aktiven Knoten. Es liegt nahe, zunächst Softwaresysteme mit einer vergleichbaren Abstraktion darauf zu untersuchen, ob sie mit diesem Modell realisierbar sind.

Datenflussbasierte Anwendungen (vgl. z. B. [Kai05]) basieren auf einem Programmiermodell aus Knoten und Kanten, wobei die Knoten Berechnungen und die Kanten Datenübergänge repräsentieren. Unter der Voraussetzung, dass der Datenfluss diskret oder zumindest geeignet diskretisierbar ist, und damit auf Nachrichten abgebildet werden kann, sind solche Systeme realisierbar. In einer datenflussbasierten Anwendung löst die Verfügbarkeit der Daten die Verarbeitung aus, dieses Verhalten ist identisch zu dem Verhalten der Komponenten im nachrichtenbasierten Modell. Eine Untergruppe der datenflussbasierten Anwendungen sind Softwaresysteme auf Basis des Pipes&Filters-Musters. Diese sind unter den selben Voraussetzungen realisierbar.

Eine Nachricht kann aber auch einem Signal entsprechen, welches eine Aktion in der empfangenden Komponenteninstanz auslöst. Hierbei müssen nicht zwangsläufig Daten durch die Nachricht übermittelt werden, alleine der Umstand des Eintreffens kann schon ausgewertet werden und ausreichend signifikant sein. In realen Systemen wird die Nachricht allerdings zumindest Daten über die Art des Ereignisses transportieren. Da hier keine Berechnungen auf einen Datenstrom innerhalb der Anwendung angewandt werden, handelt es sich bei diesen ereignisbasierten Systemen konzeptionell um eine andere Gruppe von Softwaresystemen als bei den zuvor angesprochenen datenflussbasierten Anwendungen. Zu berücksichtigen ist, dass die vorgenommene prototypische Realisierung eine asynchrone Kommunikation realisiert. Um Anwendungen auf Basis synchroner Ereignisbenachrichtigungen zu erstellen, müssten entsprechende Änderungen am Nachrichtentransport vorgenommen werden oder synchrone Übermittlungsprotokolle zwischen den Komponenten festgelegt werden.

Neben diesen eher aufgrund von Programmierparadigmen motivierten Gruppen von Software-applikationen gibt es mit den workflowbasierten Anwendungen noch eine durch die Art der Anwendung beschriebene Gruppe von Applikationen, die naheliegenderweise auf Basis der bisherigen Ergebnisse umgesetzt werden kann. Dabei wird der Übergang aus einem Zustand des Workflows in den nächsten durch das Versenden einer Nachricht abgebildet. Diese signalisiert zum einen die Aktivierung des empfangenden Prozessschritts und kann zum anderen Prozessdaten zwischen den einzelnen Schritten transportieren. Im Allgemeinen wird es sich jedoch nicht um einen Datenstrom im zuvor angesprochenen Sinne handeln.

8.9 Zusammenfassung und Fazit

Nachrichtenbasierte Anwendungen sind, wie am Anfang von Kapitel 7 schon erwähnt, keine neue Idee. In dieser Arbeit wurde die Möglichkeit einer Anwendung dieses Programmierparadigmas auf den Bereich der Endnutzerprogrammierung gezeigt. Da die zugrunde liegende Abstraktion einfach und intuitiv verständlich ist, besitzt sie großes Potenzial für eine Nutzung in diesem Umfeld. So werden beispielsweise Datenflussmodelle von Sommerville [Som01] als einfach und intuitiv verständlich bezeichnet, nach meiner Ansicht übertragen sich diese Eigenschaften direkt auf das hier diskutierte nachrichtenbasierte Modell, insbesondere bei einer entsprechenden visuellen Repräsentation.

Auf Basis des hier gezeigten Vorgehens kann ein Endnutzer nicht nur eine neue Anwendung erstellen, es ist auch möglich, bereits vorhandene Anwendungen weiter zu entwickeln. Zum einen kann dies durch die Einbindung neuer Komponenten in die Anwendung geschehen, oder zum anderen durch den Austausch vorhandener. Wie bereits kurz angesprochen kann Letzteres unter gewissen Voraussetzungen auch ohne Verwendung der Entwicklungsumgebung durch einen Austausch der Realisierungen der Komponenten geschehen. Insbesondere im Hinblick auf Fehlerbeseitigungen in einer Anwendung ist dieser letzte Punkt interessant. Die durch den Einsatz der Komponenten erreichte Kapselung und die lose Kopplung, welche sich durch die nachrichtenbasierte Kommunikation ergibt, ermöglichen eine lokale Fehlerbeseitigung durch den simplen Austausch einer Komponente.

Ein Nachteil des hier diskutierten Ansatzes entspringt aus der Black-Box-Eigenschaft der verwendeten Komponenten. Der Endnutzer hat im Allgemeinen keine Möglichkeit, deren Korrektheit zu prüfen. Vergleichbares gilt für die Realisierung der Kommunikation zwischen den Komponenteninstanzen. Hier ergeben sich durch die Möglichkeiten einer Aufzeichnung und Auswertung der Nachrichten auch Bedrohungspotentiale im Bereich des Datenschutzes. Zwar kann eine Protokollierung von Nutzeraktionen prinzipiell in jeder Anwendung erfolgen; der Nachrichtenvermittler als zentraler Punkt in der Kommunikation der Komponenten vereinfacht solche Unternehmungen jedoch. Die Korrektheit und Vertrauenswürdigkeit einer Anwendung für einen Endanwender nachprüfbar zu machen, ist eine weitere Herausforderung für zukünftige Forschungsarbeiten.

Im nun folgenden Kapitel 9 werden einige Anwendungen des nachrichtenbasierten Programmiermodells auf konkrete Anwendungsfamilien gezeigt.

9 Anwendungen des nachrichtenbasierten Programmiermodells

„ “Forty-two“, said Deep Thought, with infinite majesty and calm. - The answer to the Great Question of Life, The Universe, and Everything.“
(„Zweiundvierzig“ sagte Deep Thought mit unendlicher Würde und Ruhe. - Die Antwort auf die große Frage nach dem Leben, dem Universum und Allem.)
Douglas Noel Adams, britischer Autor, 1952 - 2001

So weitreichend, wie die des von Douglas Adams beschriebenen Computers „Deep Thought“, sind die in diesem Kapitel vorgestellten Anwendungsbeispiele für das nachrichtenbasierte Programmiermodell zwar nicht; dennoch sind sie ausreichend, um einen Eindruck von den Möglichkeiten zu vermitteln und die Anwendbarkeit der bisherigen Ergebnisse in konkreten Problemfeldern zu zeigen. Zu diesem Zweck werden zunächst die notwendigen Schritte zur Anpassung auf eine Anwendungsfamilie erläutert, dann werden zwei konkrete Beispiele vorgestellt. An einem dieser Beispiele wird ein Szenario der Nutzung der Entwicklungsumgebung beschrieben. Eine Reihe weiterer Anwendungsfelder, in denen sich die Ergebnisse dieser Arbeit sinnvoll einsetzen lassen, werden zum Abschluss des Kapitels angesprochen.

9.1 Einleitung

In diesem Kapitel sollen einige Anwendungen des nachrichtenbasierten Programmiermodells vorgestellt werden, welches in Kapitel 7 konzeptuell betrachtet wurde. Die Anwendung der im vorigen Kapitel 8 vorgenommen Realisierung des Programmiermodells auf eine konkrete Anwendungsfamilie, also das Bereitstellen von Komponenten und Sprachbeschreibung, dient auch dazu, nach der grundsätzlichen Realisierbarkeit der bisherigen Überlegungen auf Basis existierender Technologien auch die Anwendbarkeit auf eine konkrete Problemdomäne zu zeigen. Zunächst wird das allgemeine Vorgehen hierzu erläutert, diese Darstellung ist auch als Leitfaden für die Erstellung anderer Anwendungsfamilien auf dieser Basis zu sehen.

In Abb. 9.1 ist die Einordnung der Abhandlungen dieses Kapitels der Ausarbeitung in den Gesamtzusammenhang anhand der in Kapitel 1 (siehe Seite 12) erstmals gezeigten Übersicht dargestellt. Nachdem die in den vorangegangenen Kapiteln dargestellten Ergebnisse auf eine Anwendung durch Softwareenwickler ausgerichtet sind und der Realisierung von Lösungen zur Verwendung in der Endnutzerprogrammierung dienen, wird in diesem Kapitel erstmals die Ebene der Nutzung durch einen Endanwender betrachtet.

Nach einer kurzen Darstellung der zur Realisierung einer Anwendungsfamilie notwendigen Schritte (siehe Abschnitt 9.2) werden zwei prototypisch umgesetzte Beispiele für Anwendungsfamilien dargestellt. In diesem Zusammenhang wird auch die Nutzung der Entwicklungsumgebung beim Erstellen eines konkreten Exemplars einer solchen Anwendungsfamilie vorgestellt. Hier ist anzumerken, dass die in Abschnitt 9.2 erläuterten Prozessschritte durch einen Softwareentwickler durchzuführen sind. Sie stellen das Bindeglied zwischen den Ergebnissen aus Kapitel 8 und den konkreten Anwendungsfamilien in diesem Kapitel dar. Erst das Ergebnis dieser Schritte – dies entspricht den in den Unterkapiteln 9.3 und 9.4 erläuterten Anwendungsbeispielen – ist zur Verwendung in der Endnutzerprogrammierung geeignet.

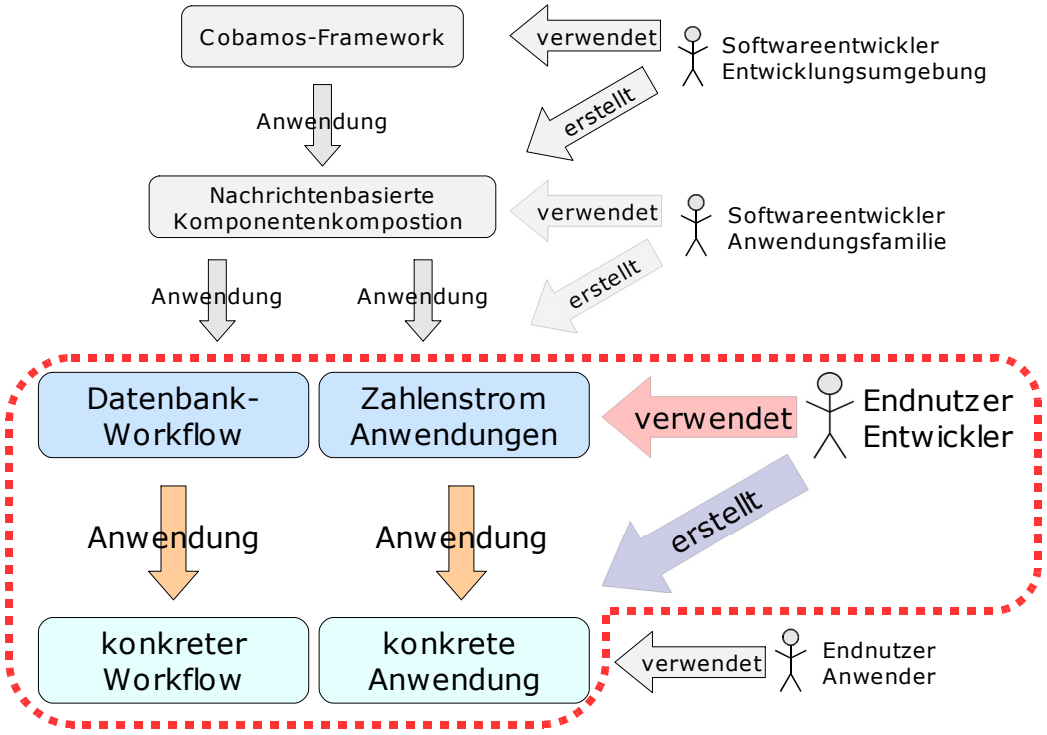


Abb. 9.1: Einordnung der Betrachtungen dieses Kapitels in den Gesamtzusammenhang

9.2 Vorgehen zur Realisierung einer Anwendungsfamile

Der erste Schritt einer solchen Entwicklung ist – wie im Falle eines jeden Projektes zur Erstellung von Software – die Beschreibung der Problemstellung. Hierbei muss geprüft werden, ob sich das Problem geeignet mit den vorhandenen Ansätzen lösen lässt, ob eventuell Ergänzungen und Weiterentwicklungen notwendig sind oder ob es sinnvoller ist, einen völlig anderen Ansatz – beispielsweise unter Verwendung eines anderen Programmiermodells – zu wählen. In Abschnitt 8.8 wurde eine kurze Betrachtung möglicher Einsatzbereiche für das nachrichtenbasierte Programmiermodell angestellt, diese bietet eine Unterstützung für eine solche Entscheidung.

Der Prozess, das Programmiermodell auf eine Anwendungsfamilie anzupassen, kann in eine Reihe von Einzelschritten unterteilt werden. Einige der Aufgaben können in der Reihenfolge vertauscht oder auch parallel ausgeführt werden.

Zunächst muss der geplante Gesamtumfang der Anwendungsfamilie bezüglich der Funktionalität, welche später verfügbar sein soll, abgesteckt werden. Durch die Verwendung der Komponenten als Anwendungsbasis und die lose Kopplung zwischen den einzelnen Anwendungsbausteinen kann die Umsetzung der Gesamtfunktionalität in mehreren Evolutionsschritten erfolgen. Für eine erste Version ist es ausreichend, die in jedem Fall notwendige Basisfunktionalität zur Verfügung zu stellen; Erweiterungen können in Form zusätzlicher Komponenten in darauffolgenden Schritten hinzugefügt werden. Damit eine solche Entwicklungsform erfolgreich sein kann, muss das Nachrichtenformat für die Anwendungsfamilie sorgfältig entworfen werden, um auch solchen Anforderungen an die Kommunikation zwischen den Komponenten genügen zu können, die sich erst in späteren Ausbaustufen des Systems ergeben. Eine Änderung am Nachrichtenformat, die inkompatibel zu den in den ersten Versionsschritten realisierten Komponenten ist, erfordert eine Anpassung aller bis zu diesem Zeitpunkt vorhandenen Komponenten und sollte aus diesem Grund vermieden werden.

Wenn ein funktionaler Rahmen für die Anwendungsfamilie festgelegt ist, müssen die Verantwortlichkeiten der einzelnen Komponenten bestimmt werden. Nun kann auch eine Priorisierung der Funktionalitäten als Grundlage des eben angesprochenen evolutionären Entwicklungsprozesses für die Anwendungsfamilie vorgenommen werden. Bei der Festlegung der Verantwortlichkeit einer Komponente ist auf eine sinnvolle Größe im Anwendungsbereich zu achten (vgl. Komponentendefinition in Kapitel 3). Es ist eine Entwurfsentscheidung, auf welcher Abstraktionsebene die Anwendungen durch einen Endnutzer beschrieben werden sollen. Von dieser Entscheidung hängt die zu wählende Granularität der zur Modellierung der Anwendungen benötigten Komponenten ab. Es dürfte in den meisten Fällen sinnvoll sein, dass sich die gewählte Abstraktionsebene über die Gesamtheit der Bausteine nicht zu sehr unterscheidet. Ausnahmen hiervon können sich ergeben, wenn spezielle Komponenten einer niedrigeren Abstraktionsebene für abgegrenzte Ausschnitte der Anwendung bereitgestellt werden. Ein Spezialist für diesen Teilbereich kann dann auf dieser niedrigeren Ebene eine angepasstere Variante der Anwendung beschreiben. In allen Fällen sollte eine Komponente eine abgeschlossene Aufgabe erfüllen oder eine in sich geschlossene Abstraktion des modellierten Anwendungsbereiches darstellen.

Der Entwurf der Komponenten ist eng mit der Festlegung der Sprache für die Nachrichten verknüpft, da hierbei festgelegt wird, welche Informationen ausgetauscht werden müssen. Mit der Festlegung der Nachrichtensprache und welche der Komponenten bzw. ihrer Ein- und Ausgangsports welche Sprache unterstützen müssen, kann – wenn diese nötig ist – auch der Entwurf einer Prüffunktionalität (vgl. Abschnitt 8.4 zur Realisierung von Bedingungen und Modellbeschränkungen) erfolgen. An dieser Stelle kann auch die Realisierung einer Komponente zur Prüfung semantischer Beschränkungen für das Modell erfolgen. Für die Bedingungen, die sich direkt durch die nachrichtenbasierte Komponentenkomposition ergeben, steht eine solche zur Verfügung.

Wenn diese Teile der Anpassung auf eine Anwendungsfamilie erfolgt sind, ist die eigentliche Funktionalität für die späteren Anwendungen realisierbar. Die restlichen Schritte dienen der Integration in die IDE und der Konsistenzsicherung. Für die visuelle Darstellung in der Graphsicht müssen entsprechende Klassen realisiert werden, dies entspricht für einfache Visualisierungen jedoch einem Konfigurieren der vorhandenen Schablonen. Daneben ist die XML-Beschreibung der Komponenten zu erstellen, damit diese durch die Entwicklungsumgebung respektive das nachrichtenbasierte Programmiermodell nutzbar sind. In Folgeprojekten zu dieser Arbeit könnten einfache Visualisierungsklassen unter Verwendung der XML-Beschreibung generiert werden, dabei kann der bereits vorgestellte Anwendungsgenerator (siehe Abschnitt 8.7.2, Generator für die Startanwendung) zum Einsatz kommen.

Die im Folgenden beschriebenen Anwendungsbeispiele sind so gewählt, dass sie die Bandbreite der Möglichkeiten verdeutlichen können. Die Implementationen sind prototypischer Natur.

9.3 Anwendungsbeispiel Operationen auf Zahlenfolgen

9.3.1 Überblick

Dieses zuerst vorgestellte Anwendungsbeispiel gehört in den Bereich der datenstombasierten Anwendungen. Die Daten sind ganze Zahlen, und es stehen eine Reihe von Komponenten zur Manipulation solcher Zahlenströme zur Verfügung. Dabei ist es möglich, dass innerhalb einer Anwendung zu einem Zeitpunkt mehrere Datenströme existieren und – da im Falle einer datenstromorientierten Applikation die Verfügbarkeit von Daten die Verarbeitung auslöst – auch mehrere Komponenteninstanzen aktiv sind.

Neben einer Generatorkomponente, die zum Erzeugen von Zahlenströmen dient, existieren in der prototypischen Realisierung noch eine Filterkomponente, eine Komponente zum Berechnen von Partialsummen, eine Komponente für zweistellige Rechenoperationen und eine Komponente zum Anzeigen von Zahlenströmen. Die Generatorkomponente kann Zahlenfolgen entweder statisch, zufällig aus einem Intervall oder auf Basis einer Funktion erzeugen, sie besitzt nur einen Ausgang, über den sie die entsprechenden Nachrichten versendet. Der Filter besitzt zwei Eingänge und einen Ausgang, einer der Eingänge fungiert bildlich gesehen als Ventil für den zweiten. Ist der Wert am Ventileingang größer Null, so wird die Zahl am anderen Eingang zum Ausgang durchgereicht, anderenfalls verworfen. Die Komponente zum Berechnen der Partialsummen liefert an ihrem Ausgang immer die Summe aller bis zu diesem Zeitpunkt am Eingang eingetroffenen Zahlen. Analog liefert die Komponente für zweistellige Rechenoperationen an ihrem Ausgang das Ergebnis der Anwendung einer Rechenoperation auf die an ihren Eingängen eintreffenden Zahlen. Damit eine Berechnung durchgeführt wird, muss an beiden Eingängen eine Nachricht mit einer Zahl angekommen sein. Erhält die Komponente nur an einem Eingang eine Nachricht, so findet so lange keine Verarbeitung statt, bis der fehlende Parameter empfangen wird. Die Rechenoperation wird bei der Instanzierung der Komponente als Parameter übergeben und kann als Variablen die Werte der beiden Eingänge verwenden.

Als Einsatzbereich für diese Anwendungsfamilie ist das so genannte entdeckende Lernen geeignet, mit Ergänzung weiterer Komponenten sind auch Simulationsszenarien denkbar.

9.3.2 Anwendung der Entwicklungsumgebung zur Erstellung einer Applikation

In diesem Abschnitt wird die Nutzung der erstellten Entwicklungsumgebung zur Beschreibung einer Anwendung vorgestellt. Es wird ein typisches Nutzungsszenario ausgewählt, eine vollständige Besprechung aller möglichen Varianten würde an dieser Stelle über den Rahmen hinausgehen. Die im Folgenden dargestellten Schritte sollen von einem Endanwender durchgeführt werden.

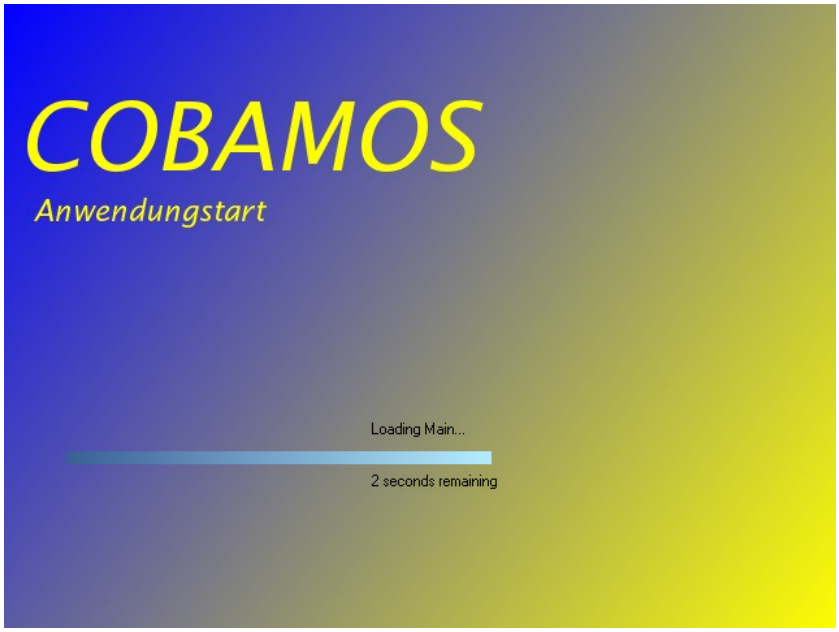


Abb. 9.2: Startbildschirm der Cobamos-Entwicklungsumgebung[79]

[79] Die Implementation des Startbildschirms basiert auf dem in [Cle03] gezeigten Beispiel.

Die Anwendung startet mit dem in Abb. 9.2 dargestellten Startbildschirm, dies dient dazu, den Nutzer über die im Hintergrund notwendigen Ladevorgänge zu informieren. Auch wenn ein solcher Startbildschirm keine eigenständige Funktionalität besitzt, erscheint der Einsatz sinnvoll, da anderenfalls für einen kurzen Zeitraum nicht zu entscheiden ist, ob die Anwendung noch korrekt arbeitet.

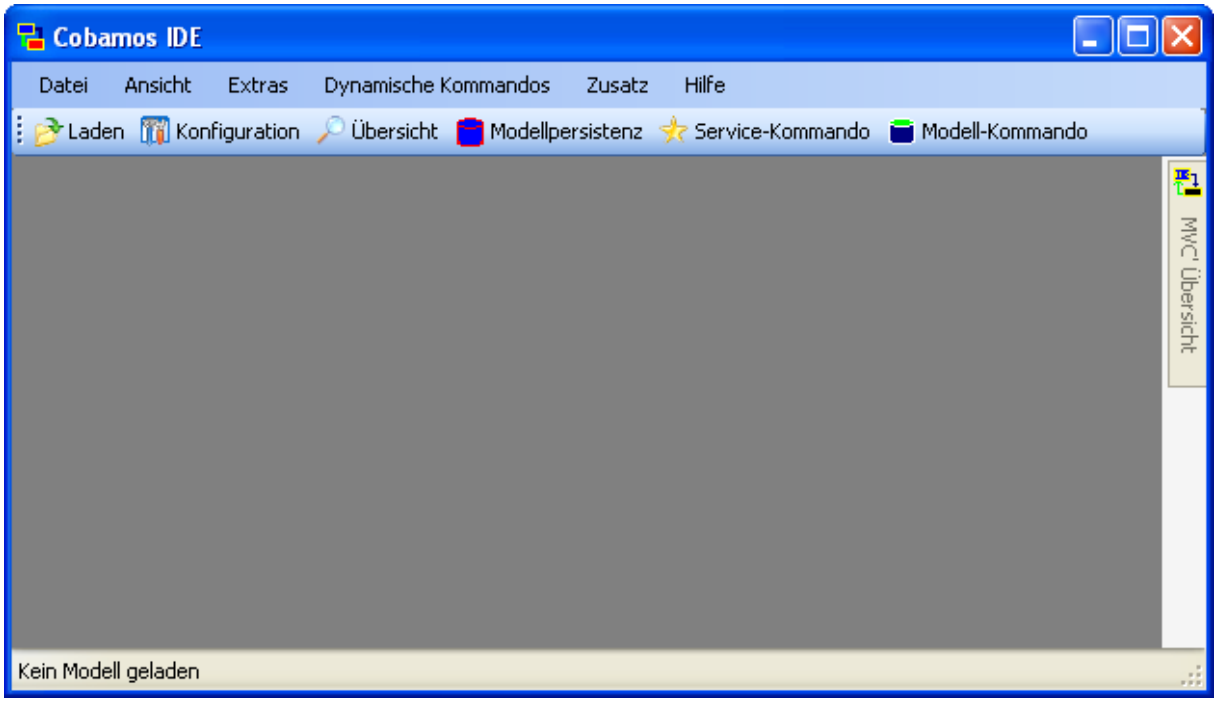


Abb. 9.3: Die Entwicklungsumgebung nach dem Start, ohne dass ein Modell geladen ist.

Nach erfolgreichem Laden präsentiert sich die Entwicklungsumgebung entweder in der in Abb. 9.3 gezeigten Form oder zeigt bereits an, dass ein Programmiermodell geladen wurde. Über Konfigurationseinstellungen kann festgelegt werden, ob beim Start der Anwendung jeweils das zuletzt verwendete Programmiermodell geladen werden soll. Zudem führt die Applikation eine Historie der letzten vier verwendeten Programmiermodelle, um einen erneuten Zugriff zu beschleunigen. Ein Programmiermodell kann auch direkt geladen werden, dies ist über den Menüpunkt „Laden“ im Menü „Extras“ oder über die Schaltfläche „Laden“ auf der Werkzeugleiste möglich. Im Rahmen dieser Arbeit wurden zwei Programmiermodelle erstellt, zum einen das in Kapitel 6 angesprochene Uhrmodell und zum anderen das Programmiermodell zur nachrichtenbasierten Komposition von Komponenten, dessen Realisierung in Kapitel 8 beschrieben ist. Hier findet das letztgenannte Verwendung, nach dem Laden steht die in Abb. 9.4 gezeigte Nutzungsoberfläche zur Verfügung. Ausgewählt ist hier die graphbasierte Sicht auf das Modell, diese wird im Folgenden zur Beschreibung der Anwendung Verwendung finden.

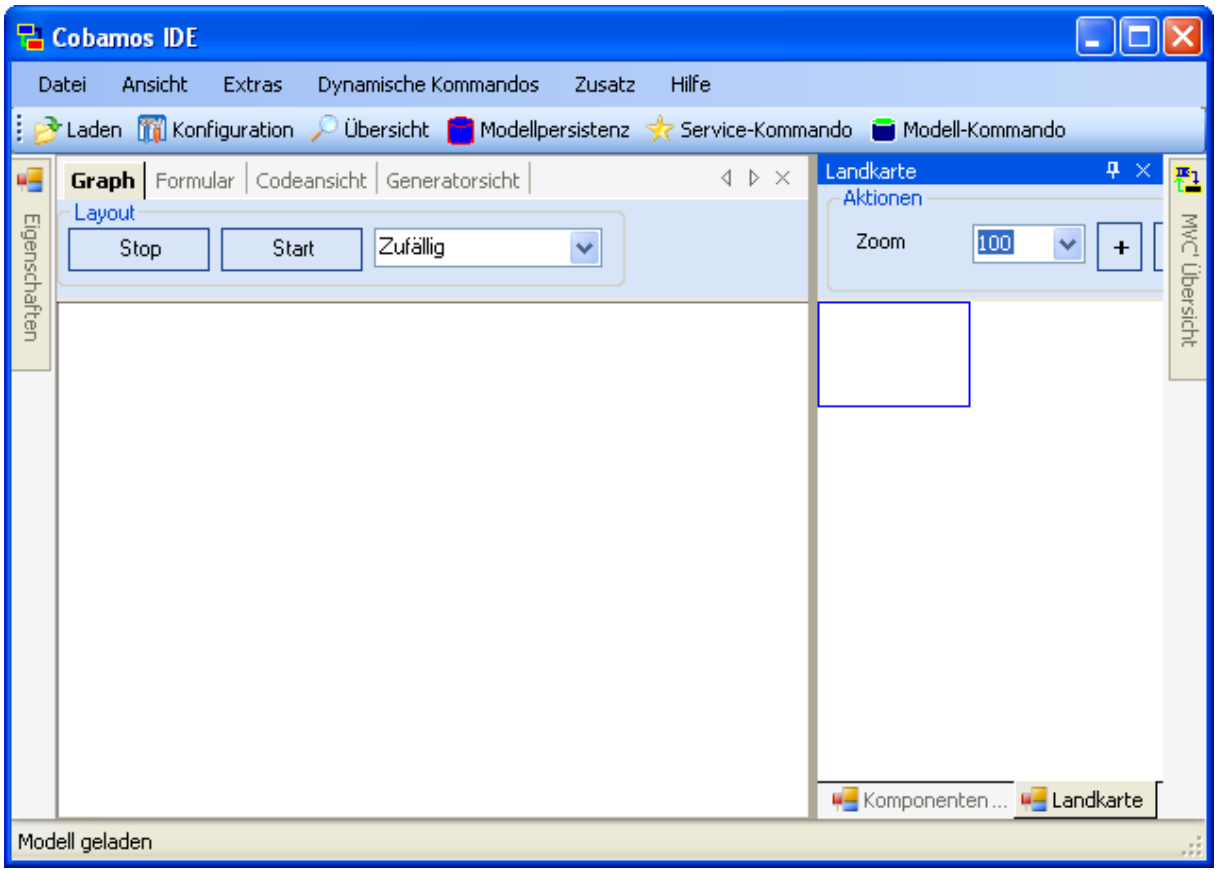


Abb. 9.4: Nach dem Laden des nachrichtenbasierten Programmiermodells

Der nächste Schritt ist die Auswahl der Komponentenliste, in welcher die verfügbaren Komponenten dargestellt sind. In Abb. 9.5 ist die ausgewählte Komponentenliste mit den verfügbaren Komponenten für die zahlenstrombasierte Anwendung zu sehen. Komponenten können per Ziehen aus dieser Liste dem Graphen hinzugefügt werden, dies entspricht der Instanzierung der Komponente in der resultierenden Anwendung. In der Abbildung wurde bereits eine Generatorkomponente zum Graphen hinzugefügt, bei diesem Vorgang wird der interne Instanzbezeichner erzeugt und angezeigt. Anstelle der intern als Bezeichner der Komponenteninstanzen verwendeten GUID lässt sich auch – in den folgenden Abbildungen ist dies zu sehen – ein menschenlesbarer Bezeichner vergeben.

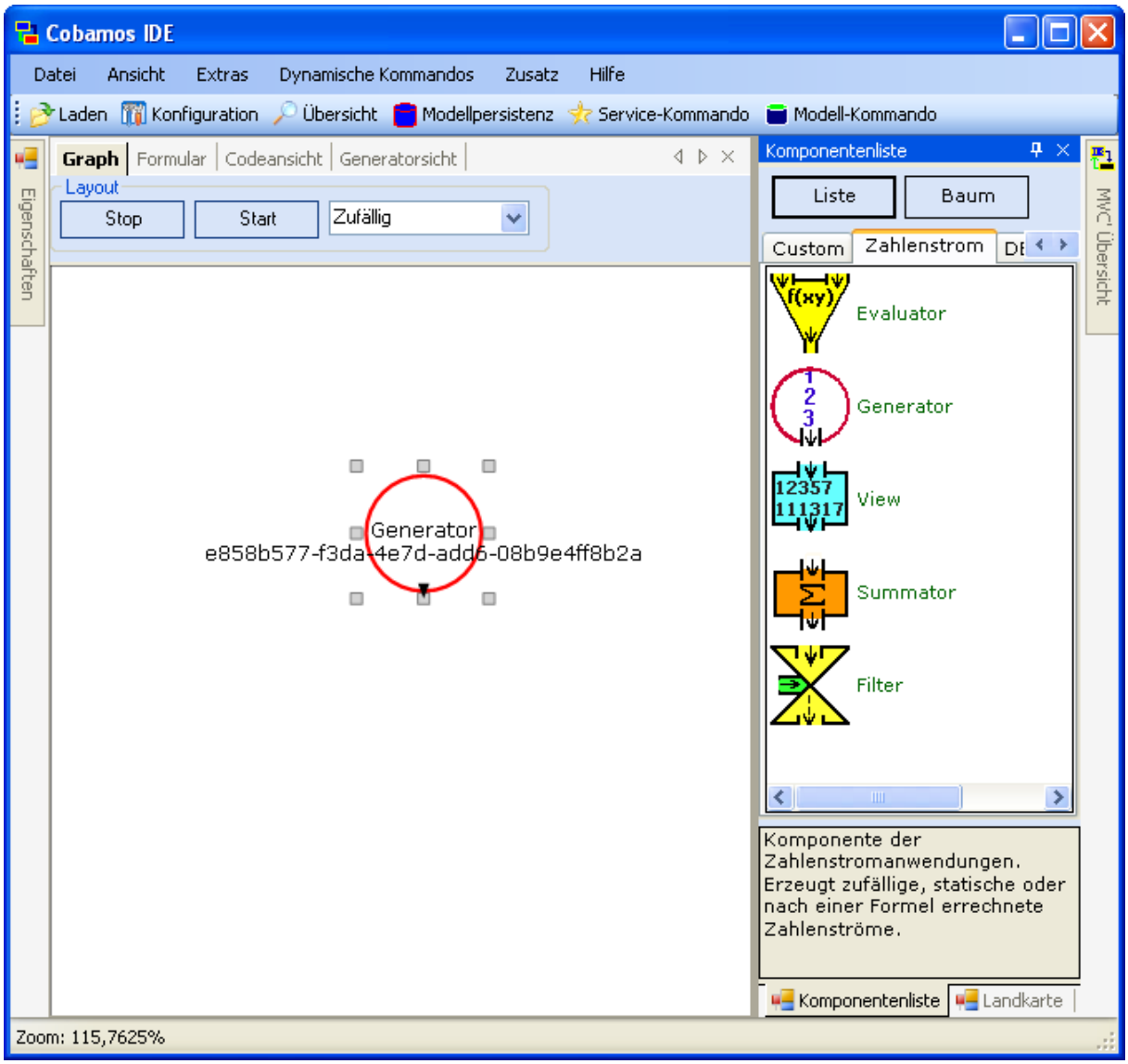


Abb. 9.5: Hinzufügen einer ersten Komponente

Im nächsten Schritt wird die eben hinzugefügte Instanz per Doppelklick ausgewählt, dadurch zeigt der Eigenschaftseditor die Informationen zu dieser an. Dies ist in Abb. 9.6 zu sehen, hier wurde bereits ein anderer Bezeichner für die Instanz („Gen_1“) vergeben. Die restlichen Eigenschaften der Gruppe Instanzinformationen legen das Verhalten zur Laufzeit der Anwendung fest. Die Formel gibt an, wie ein Wert aus dem Vorgängerwert berechnet wird. Diese Folge verwendet den Startwert als Parameter des ersten Rechenschrittes. Im Beispiel erzeugt der Generator also eine Folge der Form 1, 2, 3, 4, ... In der Abbildung ist auch zu sehen, dass in der Graphsicht der angezeigte Bezeichner aktualisiert wurde. Dieser Bezeichner ist jedoch – wie bereits erwähnt – nur eine Hilfe für den Anwender und findet intern keine Verwendung zur Identifikation der einzelnen Instanzen. Es würde daher keinen Fehler erzeugen, eine weitere Instanz mit dem identischen Bezeichner „Gen_1“ zu benennen. Wird als Bezeichner zu einem späteren Zeitpunkt eine leere Zeichenkette eingetragen, so wird wieder die intern verwendete GUID im Graphen angezeigt.

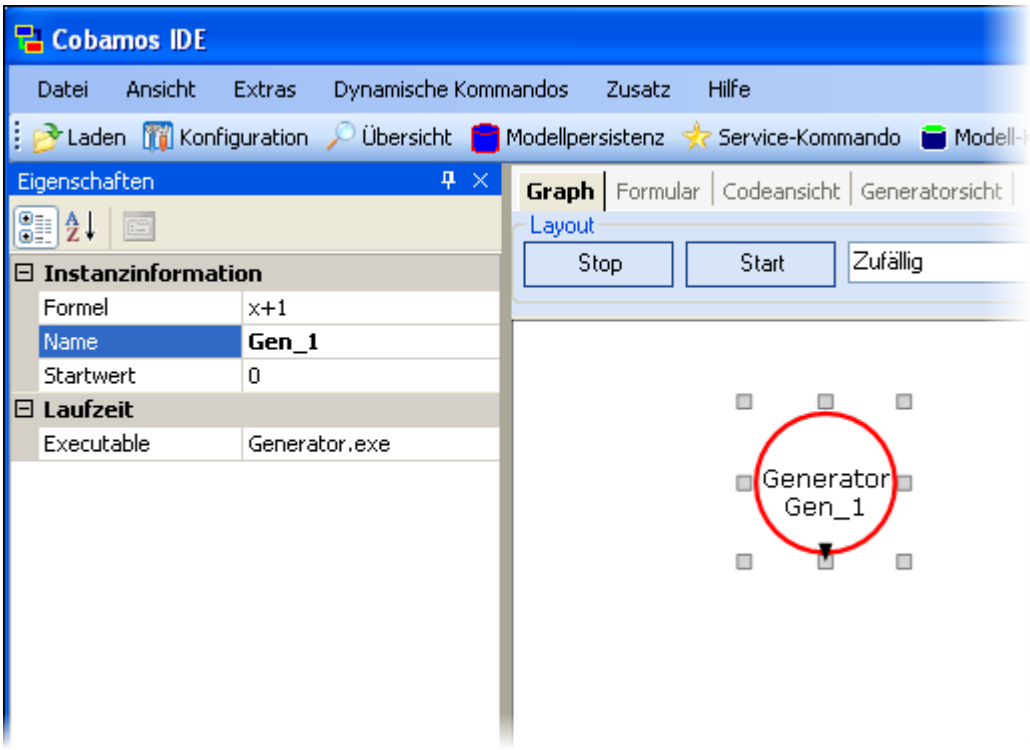


Abb. 9.6: Konfigurieren der Eigenschaften einer Komponenteninstanz

Nachdem Komponenteninstanzen zur Anwendung hinzugefügt sind, müssen die Verbindungen zwischen diesen festgelegt werden. Wie in Kapitel 8 angesprochen, ist es möglich, Modellbeschränkungen zu definieren, welche beim Anlegen von Verbindungen geprüft werden. Die Kompatibilität der verwendeten Sprachen der Komponentenports wurde bereits oben angesprochen, andere Bedingungen sind beispielsweise, dass eine Verbindung nur von einem Ausgangs- zu einem Eingangsport verlaufen darf und keine direkten reflexiven Verbindungen einer Komponente zu sich selbst zulässig sind. Die letzte Bedingung ergibt sich nicht zwingend aus dem nachrichtenbasierten Modell, ist jedoch in den hier betrachteten Anwendungsszenarien eine sinnvolle Forderung. In Abb. 9.7 ist eine Situation dargestellt, in der das Modell einen logischen Fehler – einen Verstoß gegen die formulierten Bedingungen – enthält. Die zuletzt eingetragene Verbindung der Instanz „streicheDritte“ der Filterkomponente verletzt zwei Bedingungen, dies wird dem Nutzer direkt mitgeteilt. Nach dem Bestätigen der Mitteilung wird die fehlerhafte Verbindung direkt aus dem Modell entfernt.

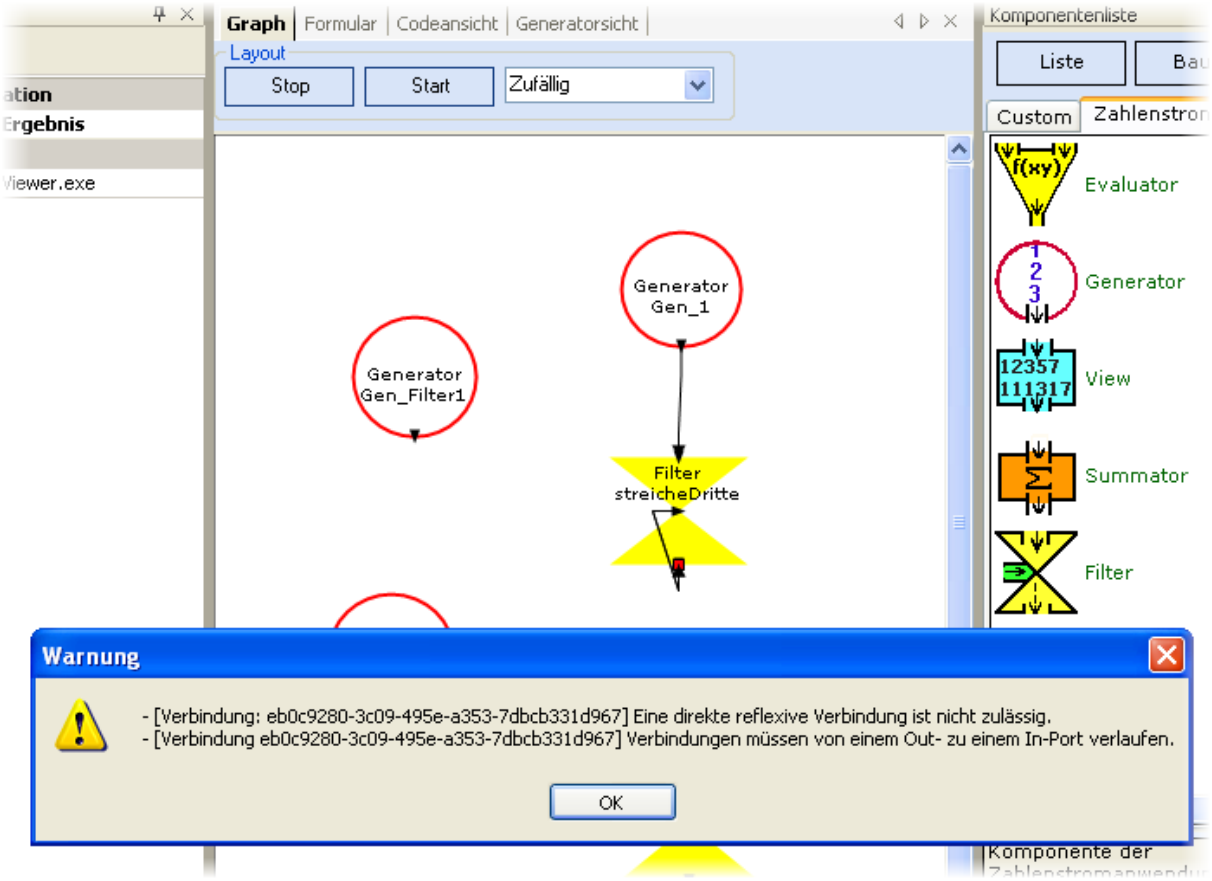


Abb. 9.7: Fehler beim Anlegen einer Verknüpfung

Es ist in der Entwicklungsumgebung möglich, Komponenten mehrerer Anwendungsfamilien parallel zur Verfügung zu stellen. Um fehlerhafte Modelle zu vermeiden, sollten entsprechende Prüfungen hinsichtlich der unterstützten Nachrichtensprachen realisiert (siehe Abschnitt 8.4) werden, wenn die Komponenten nicht kompatibel sind. In Abb. 9.8 ist eine Fehlermeldung aufgrund der Verletzung einer solchen Kompatibilitätsprüfung dargestellt. Hier wurde versucht zwei Instanzen zu verbinden, von denen die eine zu einer Komponente aus der in diesem Abschnitt besprochenen Anwendungsfamilie für zahlenstrombasierte Anwendungen gehört, die andere zu einer Komponente aus dem Anwendungsbereich der Datenbankworkflows, welcher in Abschnitt 9.4 vorgestellt wird. Die Sprachen sind in einem Fall Nachrichten, die eine ganze Zahl transportieren, im anderen handelt es sich um Nutzer- und Rechteinformationen. Auch in diesem Fall wird die fehlerhafte Beziehung nach dem Bestätigen der Mitteilung aus dem Modell entfernt.

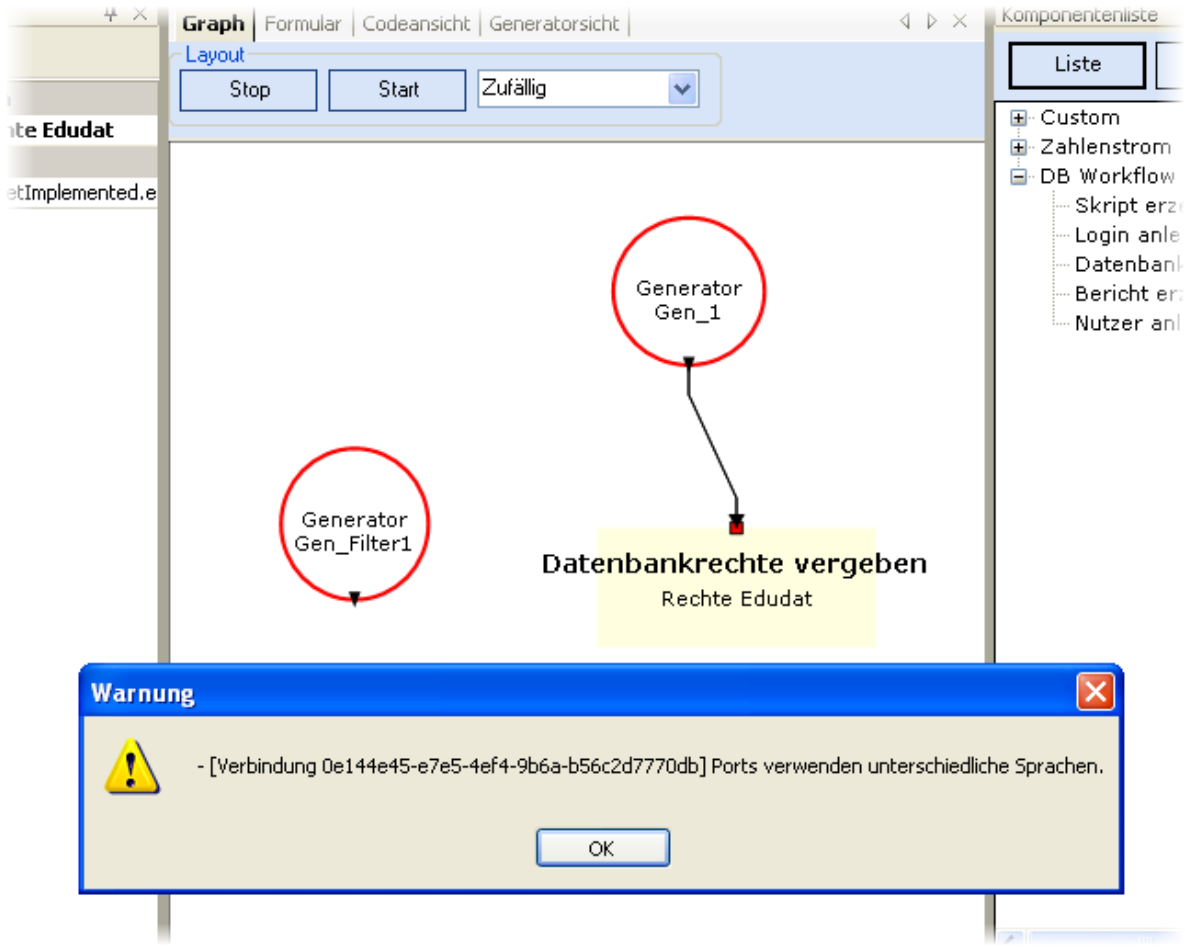


Abb. 9.8: Fehler bei Verwendung von Komponenten, deren Ports inkompatible Sprachen besitzen

Durch wiederholtes Einfügen von Komponenteninstanzen und Anlegen von Verbindungen wird der in Abb. 9.9 dargestellte Zustand erreicht, dieser Graph repräsentiert die Zielanwendung.

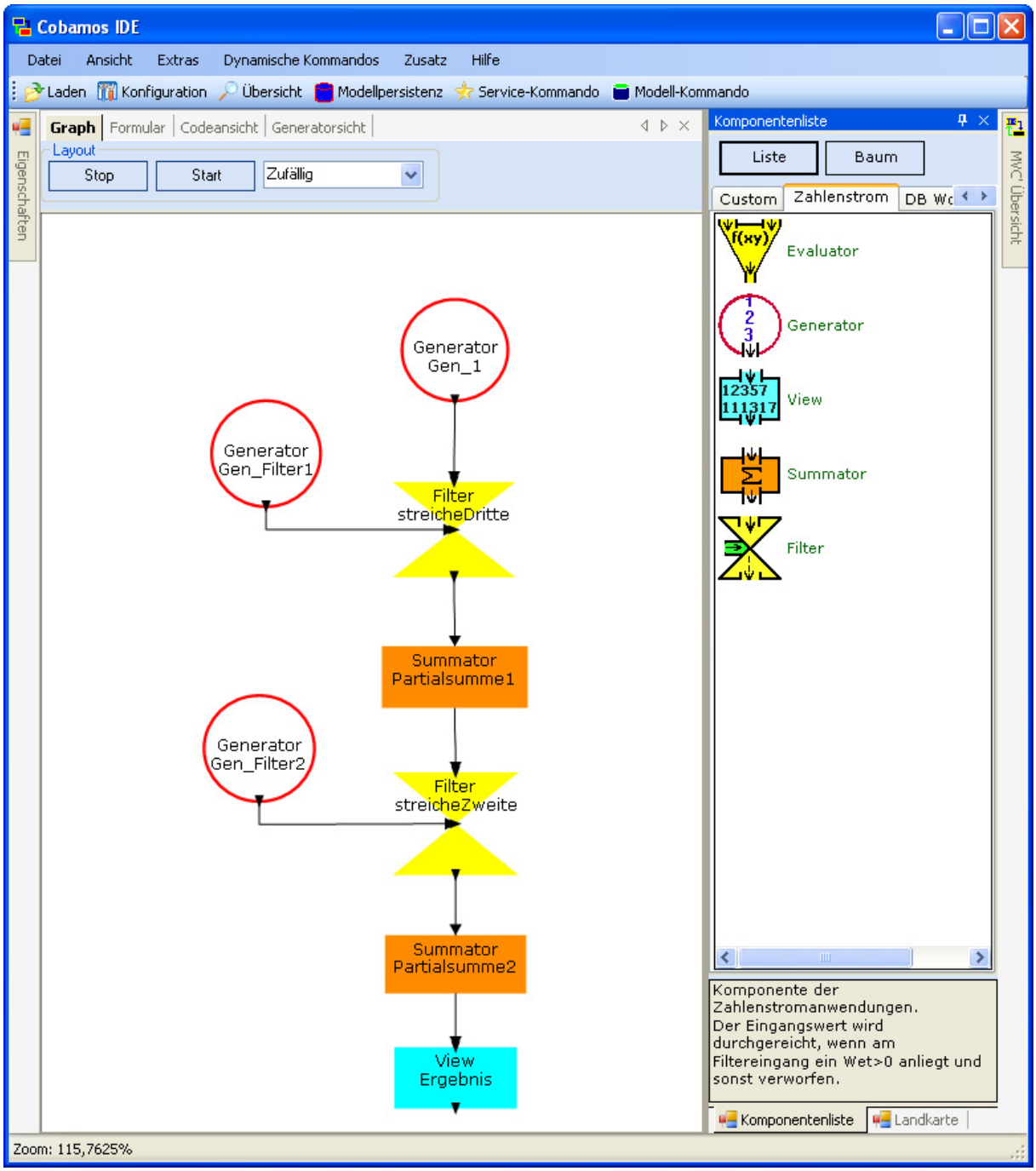


Abb. 9.9: Die fertige Applikation

Dieser Modellzustand kann nun auch in der so genannten Codeansicht – einer textuellen Darstellung – betrachtet werden. Für diese Darstellungsform sind zwei unterschiedliche Transformationen zur Umwandlung eines Modellzustands implementiert worden. Eine erzeugt ein XML-Dokument, diese Darstellung ist in Abb. 9.10 zu sehen. Dieses Format eignet sich sehr gut zur Weiterverarbeitung, ist aber beispielsweise bei der Suche nach Fehlern in der modellierten Anwendung keine besondere Erleichterung.

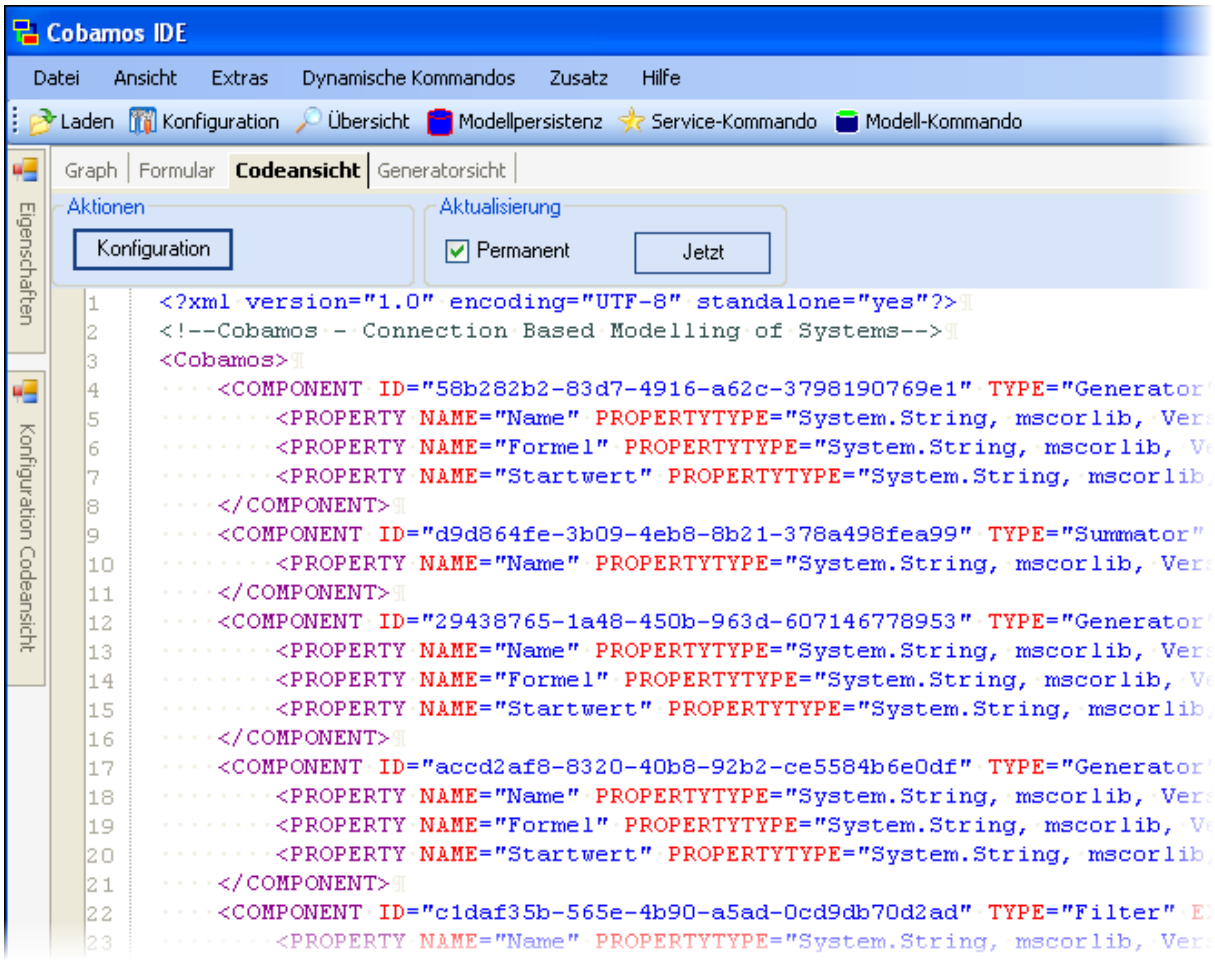


Abb. 9.10: Die XML-Darstellung der modellierten Anwendung

Die zweite textbasierte Darstellungsform für den Modellzustand ist in Abb. 9.11 zu sehen. Diese Darstellung wird als „Statistik“ bezeichnet und listet vergleichsweise übersichtlich die im Modell der Anwendung vorhandenen Komponenteninstanzen mit ihren Eigenschaftswerten und – dies ist in dem gezeigten Ausschnitt nicht zu sehen – auch die angelegten Verbindungen auf. In dieser Präsentationsform fällt nun auf, dass die Eigenschaften der beiden Generatoren, welche die „Ventileingänge“ der Filter steuern, fehlerhaft sind. Dies ist durch die Umrandungen dargestellt. Ein Filter reicht – wie am Beginn dieses Kapitels erwähnt – den Wert an seinem Eingang an den Ausgang weiter, wenn am Steuereingang ein Wert größer Null empfangen wird. Anderenfalls wird der Wert verworfen. Da die Generatoren mit den gezeigten Einstellungen aber beide die Zahlenfolge 1, 2, 3, 4, ... erzeugen würden, würden die Filter alle Werte passieren lassen. Für die Beispielanwendung sollen aber von einem der Filter jeder dritte, von dem anderen jeder zweite Wert verworfen werden.

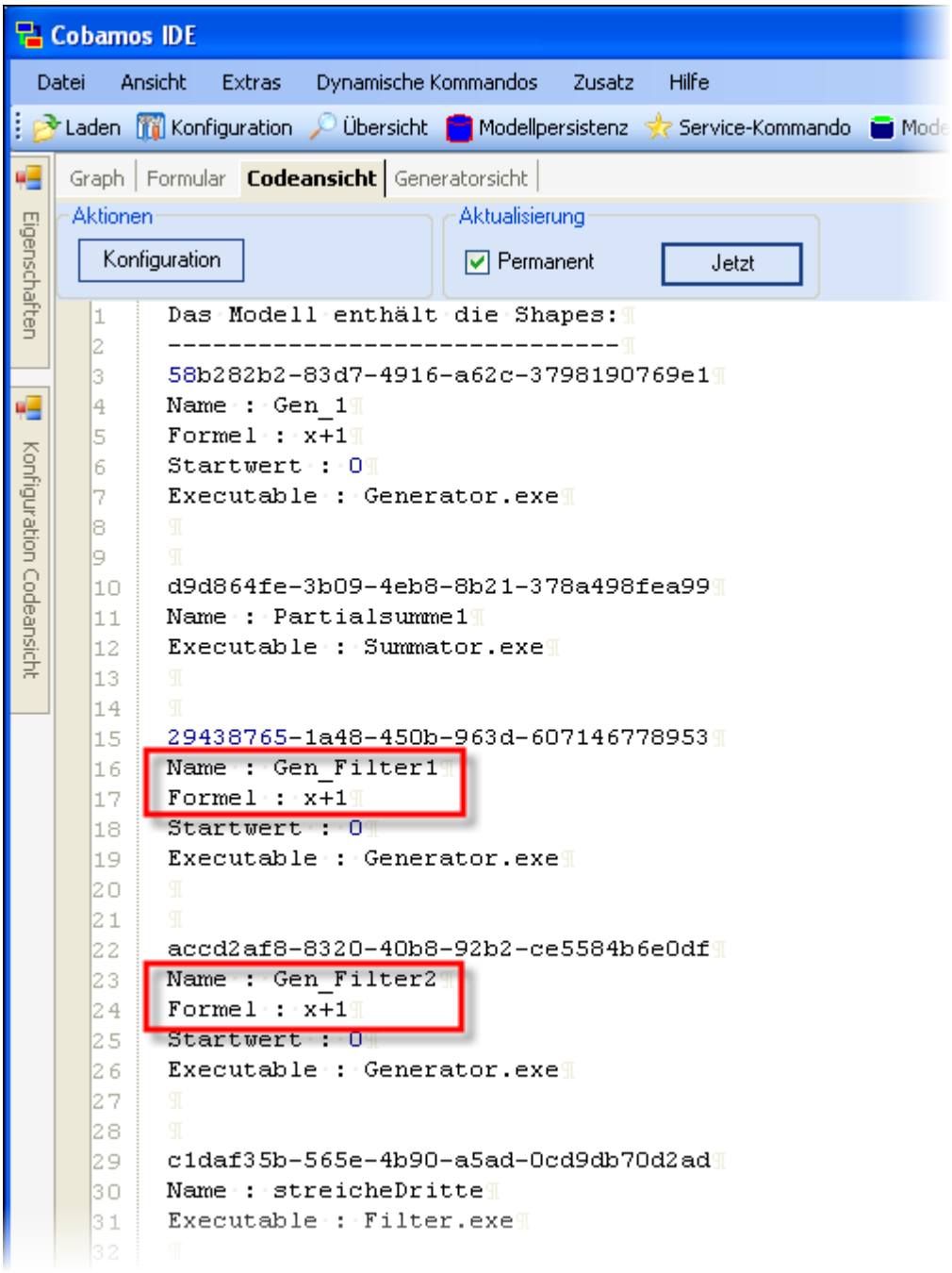


Abb. 9.11: Die „Statistik-Sicht“ auf die modellierte Anwendung, rot markiert sind fehlerhafte Eigenschaften der jeweiligen Komponenteninstanzen

Die Änderung dieser Eigenschaft ist für eine der Generatorinstanzen in Abb. 9.12 gezeigt. Durch die Verwendung der Formel `(x+1)%3` erzeugt der Generator die Folge 1, 2, 0, 1, 2, 0, ..., welche am Steuereingang des Filters gerade das Verwerfen jedes dritten Eingangswertes bewirkt.

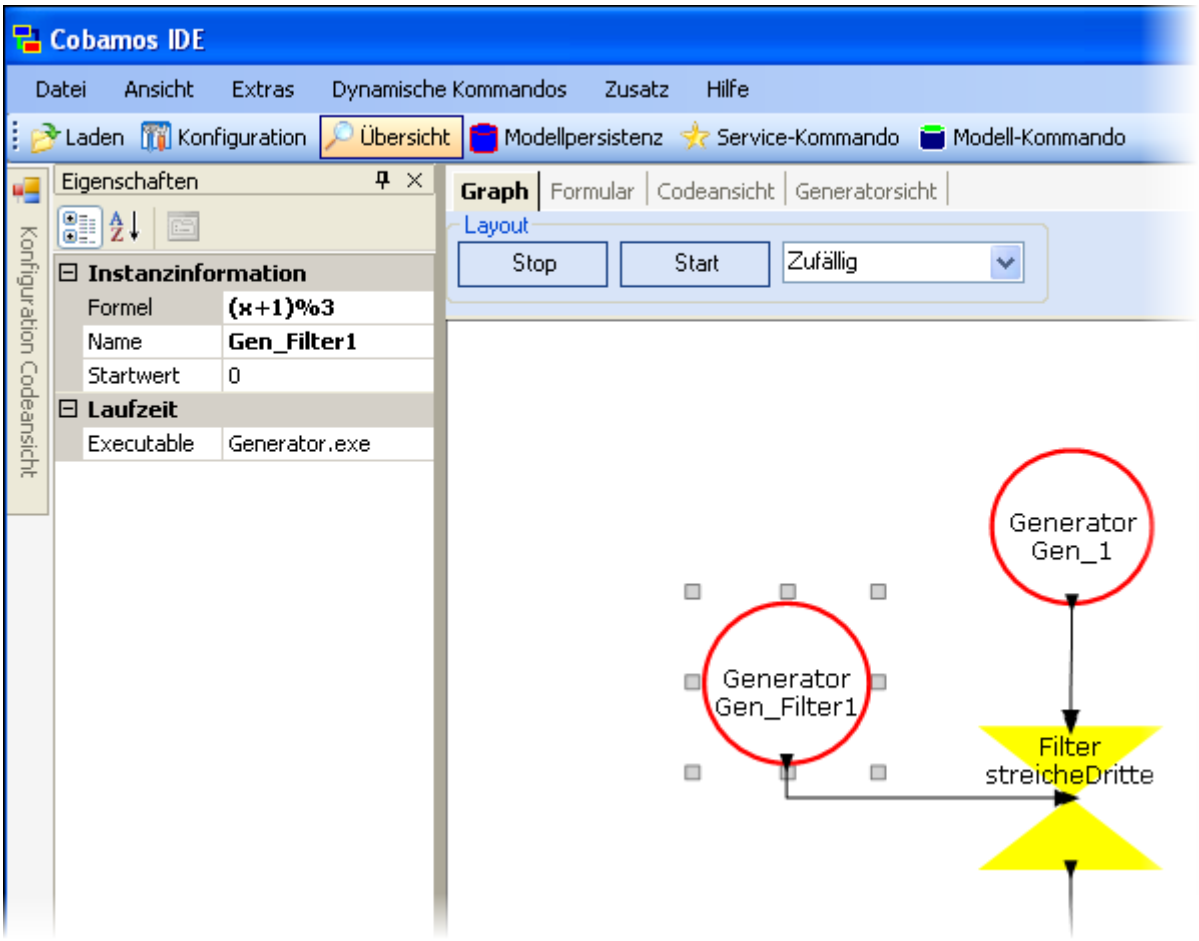


Abb. 9.12: Ändern der fehlerhaften Eigenschaftswerte

Nachdem diese Korrekturen der Eigenschaftswerte vorgenommen sind, kann die Anwendung erzeugt werden. Zu diesem Zweck ist ein Wechsel in eine andere Sicht, die Generatorsicht, nötig. Diese ist in Abb. 9.13 dargestellt und bietet einige Konfigurationsmöglichkeiten für den Generierungsschritt. Notwendig ist die Angabe eines Bezeichners für die Ausgabedatei und ein Ausgabepfad. Zudem kann eingestellt werden, wo die Komponenten zu finden sind („Komponenten Repository“); Einstellungen an dieser Stelle vorzunehmen ist jedoch nur zusammen mit der Festlegung sinnvoll, dass die Komponenten auch in das Ausgabeverzeichnis kopiert werden sollen. Bei der Erzeugung der Anwendung muss kein Zugriff auf die Komponenten möglich sein, dies ergibt sich aus dem in Kapitel 8 beschriebenen Vorgehen zur Generierung der Anwendung und der Realisierung der Komposition auf Basis des Nachrichtenaustauschs. Die generierte Anwendung startet den Cobamos-Nachrichtenvermittler, legt die Warteschlangen an und verknüpft diese und instanziert die Komponenten. Letzteres reduziert sich in der derzeitigen Realisierungsform auf das Starten des Prozesses, daher ist bei der Generierung keine Referenz auf die Komponentenrealisierungen nötig. Benötigt wird jedoch der Zugriff auf die Hilfsbibliotheken, diese Angabe ist in einer Konfigurationsdatei hinterlegt (dies gilt für alle Einstellungen dieser Sicht) und muss daher nicht bei jedem Erstellen einer Anwendung konfiguriert werden.

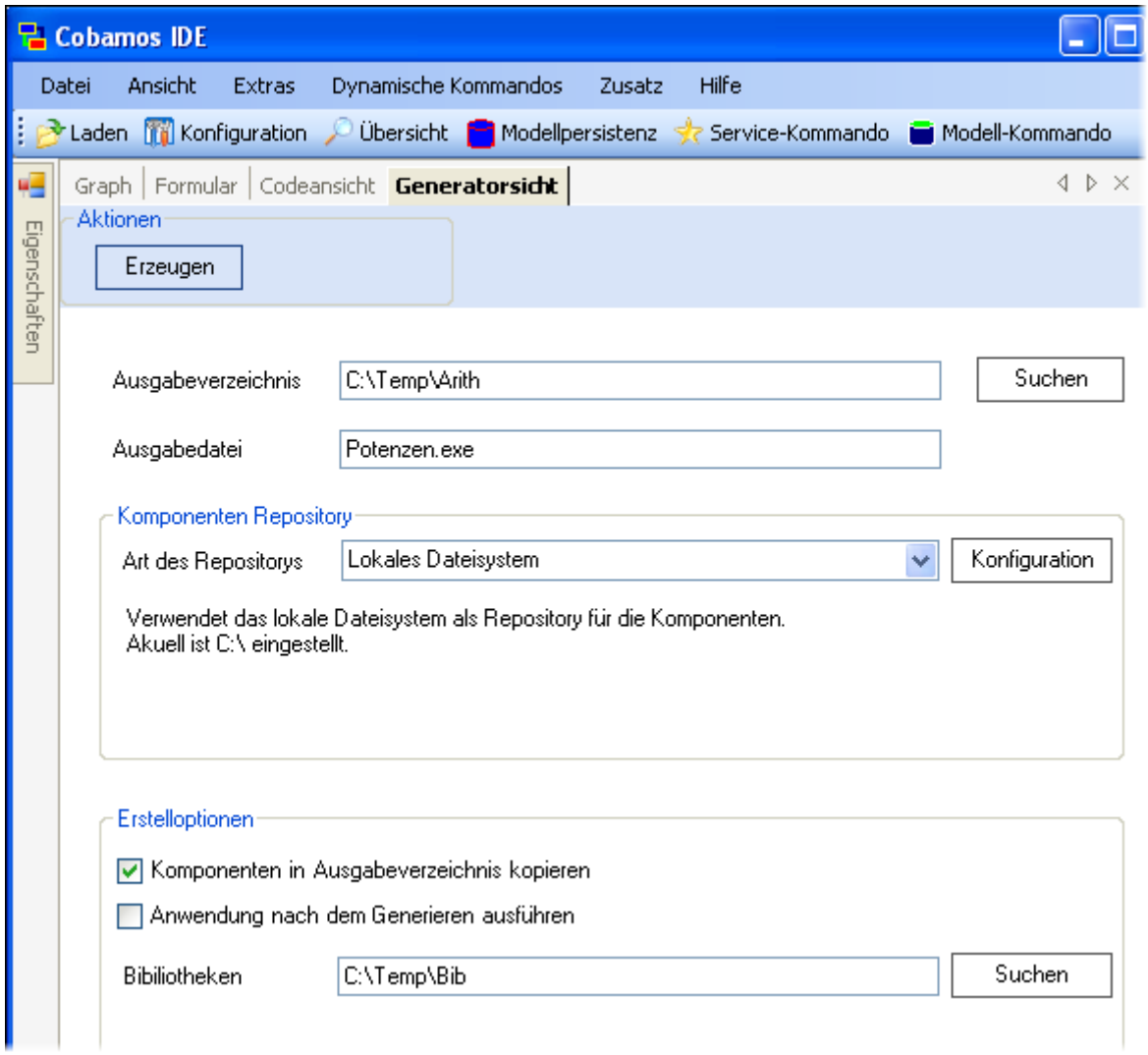


Abb. 9.13: Die Generatorsicht

Im Falle eines erfolgreichen Erzeugungsvorganges wird eine entsprechende Meldung im Ausgabefenster erzeugt, dies ist in der Abb. 9.14 zu sehen.

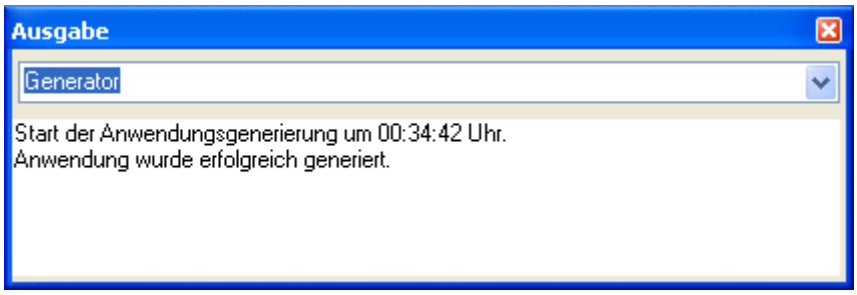


Abb. 9.14: Meldungen bei erfolgreicher Generierung der Anwendung

An dieser Stelle ist eine kritische Anmerkung notwendig. Der Fall eines nicht erfolgreichen Erzeugungsvorganges berührt ein Problem, welches im Zusammenhang mit der Nutzung von Generatoren im Einsatzbereich der Endnutzerprogrammierung auftritt (siehe auch Appendix A, hier wird dieser Punkt im Zusammenhang mit der Besprechung von Softwaregeneratoren aufgegriffen). Da

die Fehler in der Regel erst beim Übersetzen des generierten Quellcodes auftreten, sind die erzeugten Fehlermeldungen spezifisch für den Compiler der Zielsprache des Transformationsschrittes (vgl. 8.7.2). Für den Endnutzer sind diese in der Regel nicht zu verstehen und nur von geringem Nutzen. Da logische Anwendungsfehler durch die Modellbeschränkungen geprüft werden, ist im hier gezeigten Anwendungsszenario ein Fehlschlagen des Generierungsschrittes nur durch falsche Konfigurationseinstellungen möglich. Das Resultat eines solchen Fehlers ist in Abb. 9.15 zu sehen. Dieses Problem wird hier über den Ansatz, logische Fehler im Modell abzufangen und die Nachrichtenkompatibilität der kommunizierenden Komponenteninstanzen zu prüfen, nicht weiter behandelt, da es ausreichend Raum für eigene Projekte bietet. Der Punkt sollte jedoch bei einem Einsatz beachtet werden.

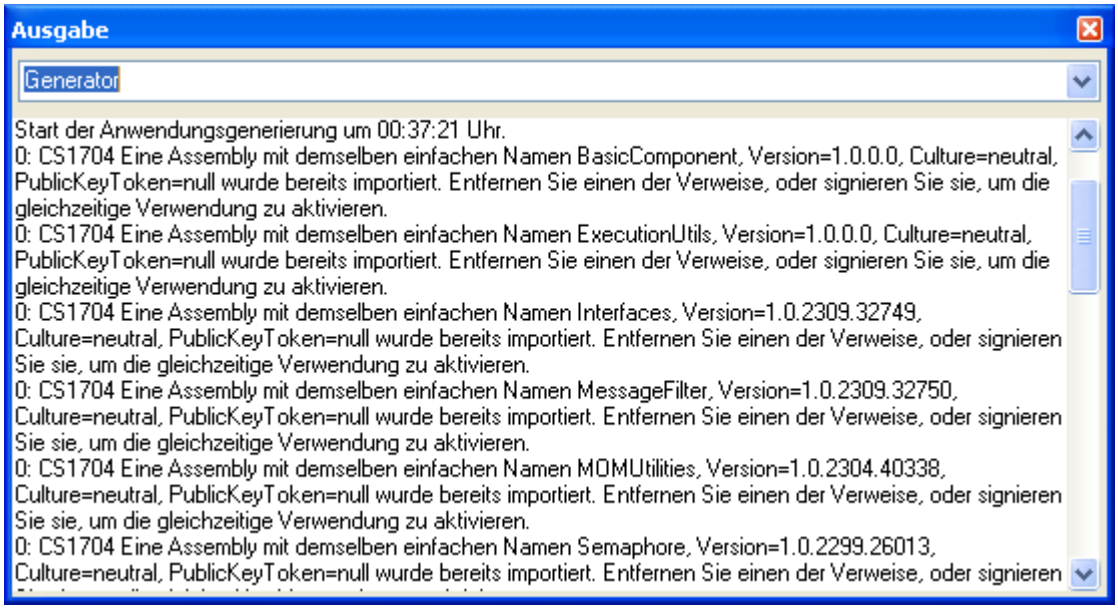


Abb. 9.15: Meldungen im Falle eines nicht erfolgreichen Generierungsschrittes

Im folgenden Abschnitt wird nun kurz auf die erzeugte Anwendung eingegangen.

9.3.3 Resultat

In Abb. 9.9 ist das modellierte Exemplar dieser Anwendungsfamilie dargestellt. In dieser Anwendung sind zur Ausführungszeit drei Datenströme zu einem Zeitpunkt aktiv. Interessant ist das Resultat der gezeigten Komponentenkomposition. Diese Anwendung berechnet dritte Potenzen auf Basis eines Verfahrens von Moessner [Moe51] (Beweis in [Per51] und [Kal89]) zur Berechnung von Potenzen natürlicher Zahlen auf Basis von Streichungen und Bildung von Partialsummen. Hier ist anzumerken, dass – wie bei der Vorstellung der Anwendungsfamilie erwähnt – als Anwendungsbereich der Lern- und Lehrbereich angedacht ist; rein zur Berechnung von Potenzen wäre diese Realisierung im Vergleich mit anderen Möglichkeiten wenig effizient. Folgende Beispielrechnung veranschaulicht das Verfahren für dritte Potenzen.

Streicht man aus der Reihe der natürlichen Zahlen jede dritte, so bleibt die Folge 1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, ... übrig. Zu dieser werden die Partialsummen gebildet, damit ergibt sich die Reihe 1, 3, 7, 12, 19, 27, 37, 48, 61, 75, 91, ..., aus welcher nun jede zweite Zahl gestrichen wird. Berechnet man zu der so erhaltenen Folge 1, 7, 19, 37, 61, 91, ... nun wieder die Partialsummen, so ergibt sich die Reihe 1, 8, 27, 64, 125, 216, ..., dabei handelt es sich um die Folge der dritten Potenzen.

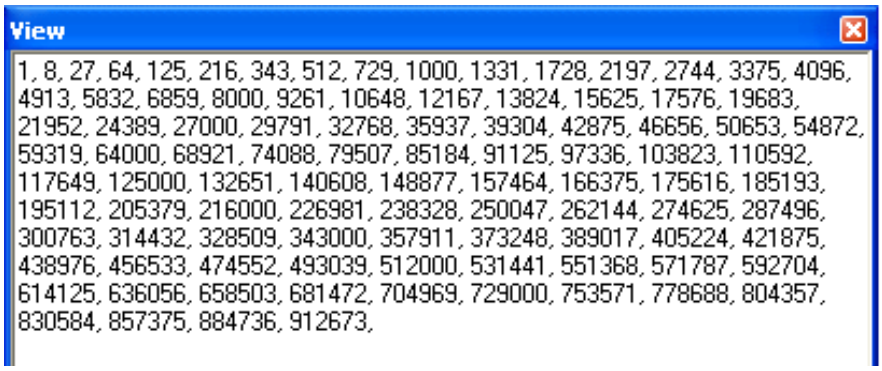


Abb. 9.16: Ergebnisse der Berechnung dritter Potenzen

In Abb. 9.16 sind die Ergebnisse eines Testlaufs mit der oben beschriebenen Anwendung zu sehen. Die Generatorkomponenten waren hierbei so konfiguriert, dass sie pro Sekunde eine Nachricht senden. Die Gesamtanwendung – also die zur Laufzeit der Anwendung vorhandenen Komponenteninstanzen – ohne das Ergebnisfenster ist in Abb. 9.17 abgebildet. Die Darstellung zeigt die Nutzungsoberflächen der aktiven Komponenteninstanzen auf dem Desktop, diese wurden dem Graph aus Abb. 9.9 entsprechend positioniert. Da die Komponenten für dieses Beispiel als eigenständige Anwendungen implementiert sind, sind die Fenster frei positionierbar und verhalten sich wie gewöhnliche Applikationen.

Dies ist auch ein Nachteil dieser Form der Anwendungsmodellierung, da für jede aktive Komponenteninstanz ein eigenes Fenster angezeigt wird. In dieser prototypischen Umsetzung wurde diese Form der Realisierung nicht zuletzt aufgrund der Nachvollziehbarkeit des Systemverhaltens gewählt. Bei der Realisierung von Anwendungen für einen produktiven Einsatz sollte sich die Anzeige auf die Nutzungsoberflächen der Komponenteninstanzen beschränken, mit denen eine Nutzerinteraktion vonnöten ist. Eine solche Verwendung wird im folgenden Abschnitt 9.4 angesprochen.

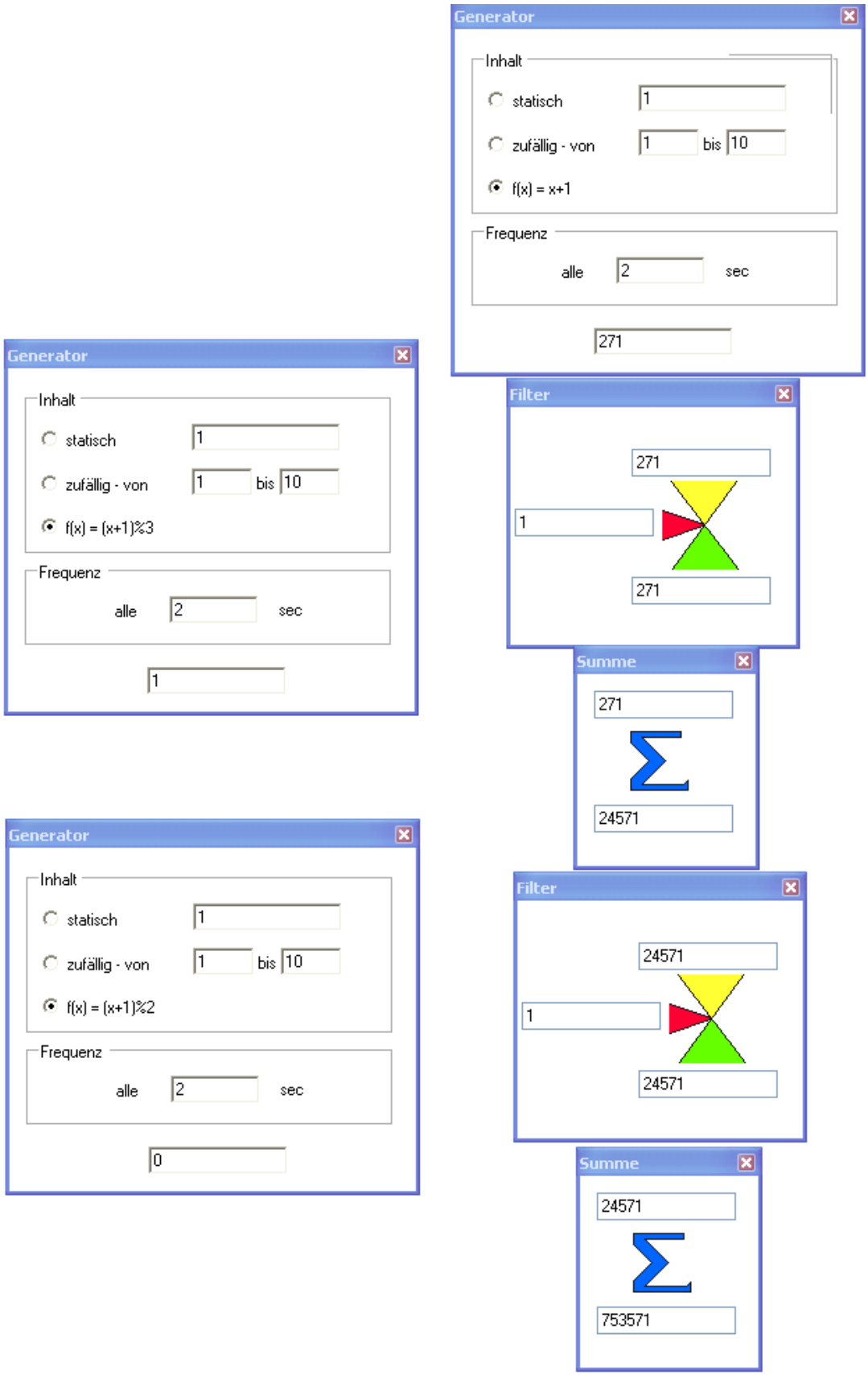


Abb. 9.17: Die aktive Anwendung zur Berechnung dritter Potenzen (ohne Ergebnisfenster, dies ist in Abb. 9.16 dargestellt)

9.4 Anwendungsbeispiel Workflow zur Datenbank Nutzerverwaltung

Ein anderes realisiertes Anwendungsbeispiel stammt aus dem Bereich der Datenbankadministration, genauer aus dem Teilbereich der Nutzerverwaltung. Dieses Beispiel wird weniger ausführlich behandelt als das vorige, die notwendigen Schritte zum Erstellen einer Anwendung sind analog zu den in Abschnitt 9.3.2 bereits beschriebenen. Die erstellbaren Applikationen sollen als Ausgangsprodukt ein Datenbankskript oder ein Anweisungsdokument zum Anlegen von Datenbankbenutzern mit den zugehörigen Rechten liefern. Es soll eine dezentrale Erstellung dieser Artefakte auch durch Anwender, die keine ausreichenden Kenntnisse zur Administration der Datenbank besitzen, möglich sein. Solche Anforderungen ergeben sich beispielsweise dann, wenn eine Fachabteilung neue Mitarbeiter einstellt. Die Informationen über den Mitarbeiterzugang müssen in einem solchen Fall an den Datenbankadministrator mit den notwendigen Details hinsichtlich Benutzerberechtigungen etc. gegeben werden. Diese Anwendungsfamilie soll hier eine Arbeitserleichterung schaffen. Anzumerken ist, dass die Implementation in erster Linie auf die Eignung als Anwendungsbeispiel ausgerichtet wurde; für einen produktiven Einsatz sind Anpassungen – beispielsweise hinsichtlich der Sicherheit – notwendig.

Dieser Anwendungsbereich wurde gewählt, da sich hier die Realisierung einer Anwendung zur Unterstützung eines Workflows zeigen lässt. Anders als im vorangegangenen Beispiel ist in diesem Szenario immer nur eine Komponenteninstanz zu einem Zeitpunkt aktiv, diese entspricht dem gerade aktuell ausgeführten Prozessschritt. Der Wechsel von einem Prozessschritt in den nächsten entspricht einem Wechsel der aktiven Komponenteninstanz, dabei muss auch die Nutzungsoberfläche wechseln. In dieser prototypischen Realisierung wird ein solcher Übergang dadurch realisiert, dass eine Komponenteninstanz ihre Nutzungsoberfläche ausblendet, sobald der entsprechende Prozessschritt verlassen wird. Die folgende Instanz im Prozess blendet ihre Nutzungsoberfläche ein, wenn sie eine Nachricht empfängt und dadurch aktiviert wird.

Während der Abarbeitung des Prozesses werden Daten über den anzulegenden Benutzer und seine Berechtigungen gesammelt. Diese Daten sind am Ende des Workflows die Eingangsdaten für den Berichts- beziehungsweise Skriptgenerator. Der Datentransport in diesem Prozess kann durch eine Nachricht modelliert werden, welche das System durchläuft und in jeder Komponenteninstanz mit zusätzlichen Informationen „angereichert“ wird. Um ein strukturiertes Datenformat zu realisieren, wird hier eine XML-basierte Nachrichtensyntax verwendet.

Die aktuelle Realisierung der Komponenten orientiert sich am Produkt Microsoft SQL Server 2005. Ein Vorteil der komponentenbasierten Umsetzung ist, dass eine Anpassung auf ein anderes Produkt durch Austauschen der produktspezifischen Komponenten geschehen kann. Eine solche ist die in Abb. 9.18 gezeigte Komponente zum Anlegen von Benutzern für einzelne Datenbanken. Da der SQL Server 2005 als Datenbankserver mehrere Datenbanken bereitstellen kann, sieht das Nutzerkonzept eine getrennte Verwaltung von Servernutzern und Datenbanknutzern vor. Dies drückt sich auch in dem in Abb. 9.19 gezeigten Workflow aus. Die Komponente in Abb. 9.18 entspricht dem zweiten Schritt. Die Liste wird mit den Datenbanken initialisiert, welche auf dem Server vorhanden sind. Die hierfür notwendige Funktionalität ist produktspezifisch. Eine Anpassung auf ein anderes Produkt könnte – wie bereits angesprochen – unter Umständen bereits durch einen Austausch der Komponentenrealisierung ohne erneutes Generieren der Applikation erfolgen.

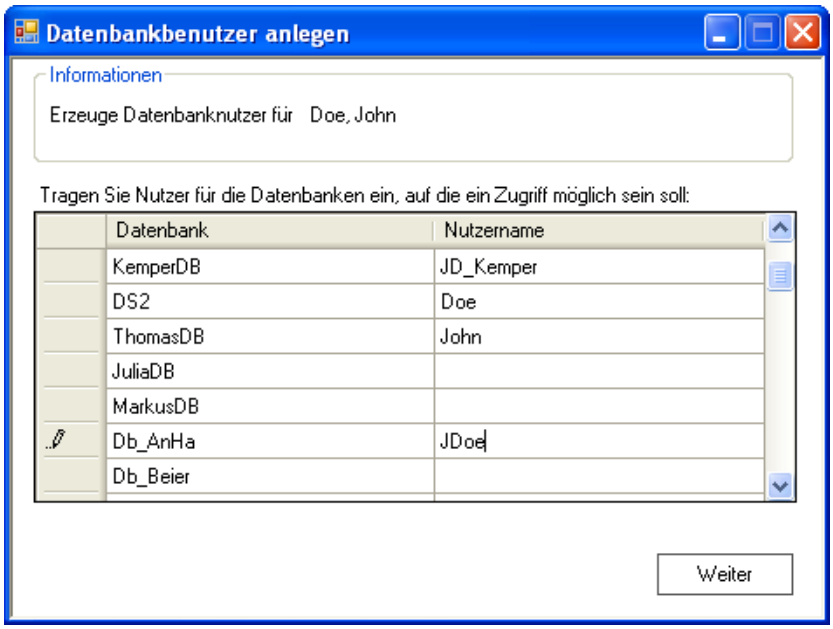


Abb. 9.18: Die Komponente zum Anlegen der Benutzer in den einzelnen Datenbanken (entspricht dem Schritt „Nutzer anlegen“ im Workflow)

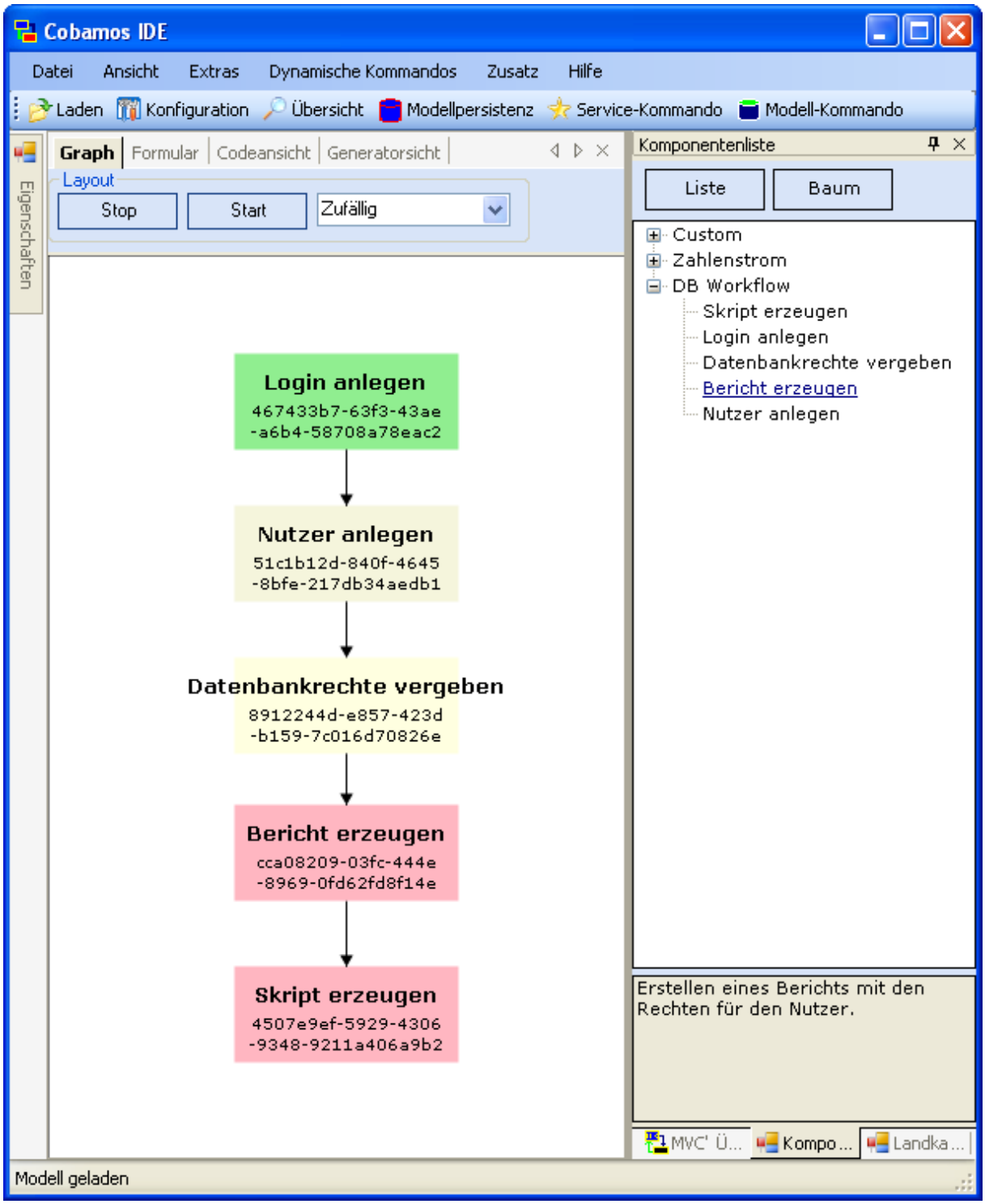


Abb. 9.19: Erstellen einer workflowbasierten Anwendung in der Cobamos-Entwicklungsumgebung

9.5 Weitere Anwendungsmöglichkeiten

In diesem Abschnitt sollen kurz einige weitere denkbare Anwendungsbereiche für das nachrichtenbasierte Programmiermodell vorgestellt werden. Diese Beispiele wurden nicht implementiert, die Diskussion an dieser Stelle soll nur einen zusätzlichen Eindruck von den Möglichkeiten vermitteln.

Realisiert man mit den Komponenten Bausteine für logische Schaltkreise und mit den Nachrichten den Signalübergang, so lässt sich auch das Beispiel, welches als Ausgangspunkt für das nachrichtenbasierte Modell zur Komponentenkomposition in Kapitel 7 diente, umsetzen. Anwendung könnte eine solche Anwendungsfamilie bei der Vermittlung von Lehrinhalten aus dem Bereich Logik und Elektronik finden.

In [VP03] wird die Anwendbarkeit von Workflowsystemen zur Realisierung von E-Learning Anwendungen gezeigt. Dieses Ergebnis legt nahe, dass auch die in den letzten Kapiteln gezeigten Ansätze geeignet sind, um Lernanwendungen zu modellieren. Vorstellbar sind Szenarien, in denen ein Lernender die für ihn interessanten Lernmodule auswählt und zu einer individuellen Lernanwendung zusammenstellt, oder in denen ein Dozent angepasste Versionen einer Lernsoftware zur Unterstützung unterschiedlicher Gruppen von Lernenden entwickeln kann. Bereitgestellt werden müssten Komponenten mit entsprechenden Lerninhalten und Aufgaben, auch Bewertungskomponenten sind denkbar, die beispielsweise vor der Nutzung bestimmter Lernkomponenten absolviert werden müssen. Ein fortgeschrittener Lerner könnte sich darauf beschränken, die Tests zu den ihm bekannten Lerninhalten zu absolvieren, während ungeübtere Lerner die Komponenten zur Vermittlung dieses Lernstoffs in ihre Anwendungen integrieren.

In der Softwareentwicklung ist das Gebiet der Anforderungsermittlung und der Erstellung von Prototypen ein mögliches Einsatzszenario. In vielen Softwareapplikationen muss ein Workflow abgebildet werden. Um diesen mit Anwendern gemeinsam zu modellieren oder einen Prototypen für diesen Workflow zu erstellen, könnte eine Reihe von prototypischen Komponenten für einzelne Workflowschritte bereitgestellt werden. Diese könnten sich auf reine Oberflächendarstellung und das Weiterreichen von Testdaten beschränken. Anwender können dann mit diesen Bausteinen einen Workflow modellieren. Die Möglichkeit, diesen zu „durchlaufen“, erleichtert die Kommunikation und die Korrektheitsprüfung.

Auch der Bereich der Sportinformatik stellt ein potenzielles Anwendungsfeld dar, hier ist insbesondere die Spielbeobachtung und Spielauswertung zu nennen. Hier könnte das vorgestellte nachrichtenbasierte Modell vorteilhaft eingesetzt werden, da es eine evolutionäre Entwicklung des Softwaresystems – auch um Bestandteile, die beim ersten Entwurf noch nicht abzusehen waren – erlaubt. In [Rei00] ist eine detailliertere Beschreibung derartiger Anwendungen zu finden.

10 Zusammenfassung und Fazit

„Beginnen können ist Stärke. Vollenden können ist Kraft.“
Laotse, Chinesischer Philosoph, 6. Jh. v. Chr.

Dieses Kapitel fasst die Ergebnisse der vorliegenden Arbeit zusammen und gibt einige Ausblicke auf mögliche weitere Projekte.

Diese Arbeit zeigte einige Ansätze auf, wie sich Komponenten im Bereich der Endnutzerprogrammierung vorteilhaft einsetzen lassen. Bei den entwickelten Konzepten und Entwürfen finden Komponenten nicht nur als Bausteine für die Anwendungen Verwendung, welche von den Endnutzern erstellt werden. Vielmehr werden in dieser Arbeit an vielen Stellen flexible Entwürfe auf Basis von Komponenten dargelegt. Im Rahmen des Entwurfs des Cobamos-Frameworks wurden Möglichkeiten zur Realisierung flexibler Entwicklungsumgebungen gezeigt, welche nicht auf ein internes Programmiermodell festgelegt sind.

Die Entwürfe dieser Arbeit sind in weiten Teilen zur Nutzung durch Softwareentwickler, als Basis für weitergehende Entwicklungen und Forschungsarbeiten in diesem Bereich gedacht. Die Flexibilität, die sich – auch dies wurde in der vorliegenden Arbeit gezeigt – durch die Verwendung komponentenbasierter Technologien in Softwareapplikationen erreichen lässt, ist beträchtlich. Eine direkte Folge solcher Überlegungen ist die Ausrichtung der Softwareerstellung auf die Realisierung konfektionierbarer Familien von Anwendungen anstelle von Einzelanwendungen.

Im Verlauf dieser Arbeit zeigten sich einige Möglichkeiten für weitergehende Entwicklungen, die aufgrund des gesteckten Rahmens nicht weiter verfolgt werden konnten. Dabei handelt es sich zum einen um Möglichkeiten zur Erleichterung der Nutzung des realisierten Frameworks bzw. des nachrichtenbasierten Programmiermodells. Interessant wären in diesem Zusammenhang Generatoren zur Erzeugung der Klassen für die visuelle Repräsentation aus den Komponentenbeschreibungen und Werkzeuge für die Generierung der Implementationen für Modellbeschränkungen auf Basis einfacher Regelbeschreibungen. Zum anderen wären Ansätze im Bereich der Nutzungsoberfläche der auf der Komponentenkomposition resultierenden Anwendung eine Bereicherung. Hier eröffnet unter Umständen die Verwendung von Markup-Sprachen zur Beschreibung visueller Oberflächen interessante Möglichkeiten. Denkbar sind Ansätze, in denen eine Komponente eine Beschreibung der von ihr benötigten Interaktionselemente enthält. Durch die Verwendung einer Markup-Sprache wäre diese Beschreibung weitgehend unabhängig von einer konkreten Realisierungsform. In einer aus mehreren Komponenten bestehenden Anwendung könnte auf Basis dieser Beschreibungen eine einheitliche Nutzungsoberfläche erzeugt werden. Kleinere Experimente unter Verwendung der Open Source XAML-Implementation (XML Application Markup Language) MyXaml [Cli06], bei denen über Attribute solche Zuordnungen vorgenommen wurden, verliefen vielversprechend.

Während der Umsetzung der vorgenommenen Implementation zeigte sich klar, dass größere Flexibilität in einer Softwareapplikation immer auch eine Erhöhung der Komplexität bedeutet. Eine Folgerung an dieser Stelle muss sein, dass eine Anwendung die Flexibilität bereitstellen sollte, die notwendig ist.

In meinen Augen ist es eine direkte und unvermeidbare Folge der fortschreitenden Computerisierung unserer Gesellschaft, dass Endanwendern verstärkt die Möglichkeit zur Entwicklung von neuen Anwendungen oder zum Anpassen vorhandener Funktionalität auf ihre individuellen Bedürfnisse

eröffnet werden muss. Aus diesem Grund halte ich den Bereich der Endnutzerprogrammierung auch für die kommenden Jahre für einen wichtigen und interessanten Forschungsbereich.

Appendix A: Generatoren und domänenspezifische Sprachen

„The best software is code you don't have to write“
(Die beste Software ist Code, den man nicht schreiben muss)
Steve Jobs, Mitgründer von Apple, Echtheit umstritten

Dieser Anhang stellt – als Exkurs zur eigentlichen Arbeit – die Themengebiete Anwendungsfamilien und Domänen sowie domänenspezifische Sprachen und Generatoren vor. Diese Begriffe finden an einigen Stellen in der vorliegenden Ausarbeitung Verwendung und werden dort – da es für die Diskussion und das Verständnis der zentralen Aspekte dieser Arbeit nicht erforderlich ist – ohne eingehende Diskussion verwendet. Da es sich jedoch um interessante Teilbereiche der Forschung handelt, welche eng an die in dieser Arbeit betrachteten angrenzen, wird dies hier nachgeholt. Dieser Anhang kann weitgehend unabhängig von der eigentlichen Ausarbeitung gelesen werden.

A.1 Einleitung und Motivation

Im Verlauf dieser Arbeit wird auf Basis des entworfenen Cobamos IDE-Frameworks ein Programmiermodell auf Komponentenbasis umgesetzt. In diesem Anhang werden einige der Konzepte und Begrifflichkeiten dargestellt, die bei diesen Erläuterungen Verwendung finden.

Hierbei ist zunächst eine Eingrenzung hinsichtlich der hiermit realisierbaren Softwareanwendungen vonnöten. In diesem Zusammenhang werden die Begriffe Domäne und Systemfamilie wichtig.

Um eine Softwareapplikation zu realisieren, ist eine – wie auch immer geartete – Beschreibung notwendig. Dies gilt genauso für die Softwareentwicklung durch professionelle Entwickler wie auch für den Bereich der Endnutzerprogrammierung. Für diesen letzten Bereich ist es jedoch wünschenswert, die Abstraktionsebene, auf der eine solche Beschreibung erfolgen muss, anzuheben. In diesem Zusammenhang bekommt das Konzept der domänenspezifischen Sprachen eine Bedeutung.

Da eine erstellte Beschreibung einer Applikation auch in das Endprodukt überführt werden muss – in der klassischen Programmierung existiert die Beschreibung in Form eines in einer Programmiersprache verfassten Quelltextes und wird compiliert – wird ein entsprechendes Konzept benötigt. Für diesen Zweck werden im Rahmen dieser Arbeit unter anderem so genannte Generatoren verwendet.

A.2 Domänen und Systemfamilien

Der Begriff **Domäne** wird in unterschiedlichen Bereichen der Informatik sehr verschieden gebraucht. Im Zusammenhang mit Datenbanken – oder genauer mit dem Relationenmodell – bezeichnet eine Domäne den Wertebereich eines Attributs (siehe beispielsweise [EN05]). Im Bereich der Netzwerktechnik werden logische Teilnetze als Domänen bezeichnet (siehe beispielsweise [Mue03]), diese Begrifflichkeit ist als „Internetdomäne“ inzwischen sogar in der Alltagssprache angekommen.

Für die Verwendung in dieser Arbeit wird eine Domäne (angelehnt an [Obj03a]) wie folgt definiert:

> *Eine Domäne ist ein Wissensbereich oder Anwendungsbereich der realen Welt, welcher durch eine Menge von zusammenhängenden Konzepten bestimmt ist. Es existiert eine Terminologie, die Anwendern aus diesem Bereich verständlich ist.*

Die in der Literatur teilweise zu findende Betrachtungsweise, eine Domäne als eine Menge von Softwaresystemen (siehe z. B. [DK98]) anzusehen oder das Wissen um den Entwurf solcher Systeme als Teil der Domäne anzunehmen, wird in dieser Arbeit nicht geteilt. Eine detaillierte Diskussion des Domänenbegriffes ist in [SCK+96] zu finden.

Eine Domäne in obigem Sinne kann unter dem Gesichtspunkt der Wiederverwendung als ein abgegrenzter Bereich betrachtet werden. Allgemein betrachtet kann Wiederverwendung einerseits entlang der Einteilung in unterschiedliche Anwendungsbereiche geschehen, dies entspräche beispielsweise einer Wiederverwendung von speziellen Datenmodellen oder von Fachlogik. Andererseits kann Wiederverwendung aber auch anwendungsbereichübergreifend auf der Ebene einer gemeinsamen Teilfunktionalität mehrerer Softwaresysteme – etwa im Sinne eines Infrastrukturdienstes – geschehen. Dies entspräche beispielsweise der Wiederverwendung einer gemeinsamen Funktionalität für das Protokollieren von Fehlern oder einer Datenbankzugriffsschicht. Ausgehend von der verbreiteten Unterteilung der Architektur eines Softwaresystems in horizontale und vertikale Schichten motivieren diese Betrachtungen die Unterscheidung **horizontaler** und **vertikaler Domänen**. Diese Begrifflichkeiten werden durch die Darstellung in Abb. A.1 verdeutlicht. Die Grafik zeigt drei Applikationen (Kredit-, Mandanten- und Produktverwaltung) in zwei vertikalen Domänen (Finanzdienstleistungen und Autoindustrie). Über mehrere vertikale Domänen hinweg erstreckt sich die horizontale Domäne „Datenbankzugriff“.

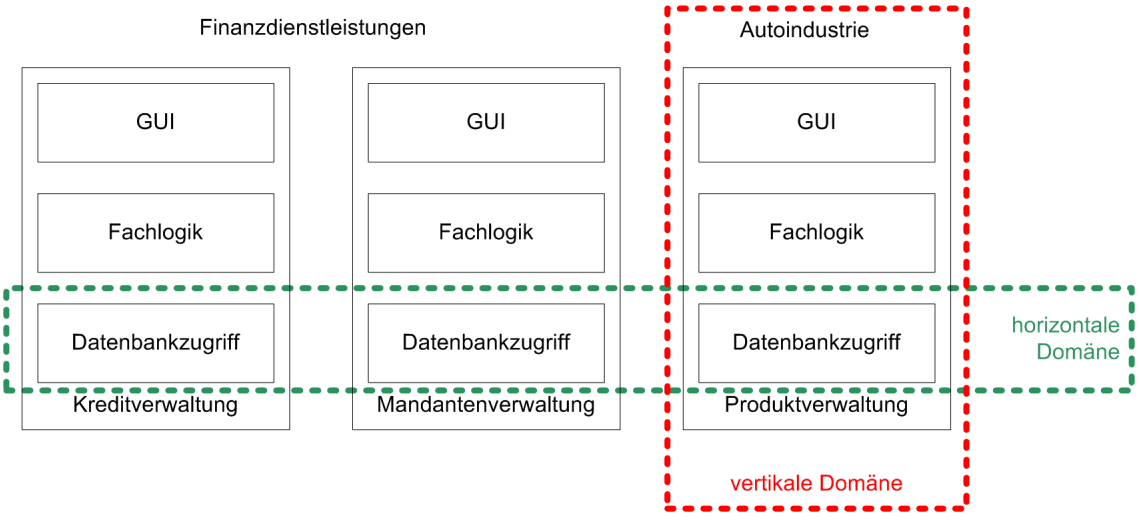


Abb. A.1: Horizontale und vertikale Domänen

Diese Begriffe sind nicht ausschließlich zu sehen, so kann die oben als horizontale Domäne angeführte Datenbankzugriffsschicht etwa aus der Sicht eines auf Datenbanksysteme spezialisierten Herstellers als zu einer vertikalen Domäne gehörig gesehen werden. Eine vertikale Domäne kann in der Regel in mehrere horizontale Domänen unterteilt werden.

Innerhalb einer vertikalen Domäne (beispielsweise Flugzeugbau, Finanzdienstleistungen) ist immer noch ein sehr breites Spektrum von Anwendungen denkbar; die Komplexität eines Systems zur Endnutzerprogrammierung für den gesamten Bereich wäre immer noch recht hoch. Um diese Komplexität weiter zu reduzieren, kann die Einschränkung auf **Familien von Softwaresystemen** betrachtet werden. Eine Familie von Softwaresystemen wird dabei wie folgt definiert:

> *Eine Familie von Softwaresystemen (auch Systemfamilie oder Anwendungsfamilie) ist eine Menge von Softwaresystemen, die genug gemeinsame Eigenschaften besitzen, so dass sie auf Basis einer gemeinsamen Menge von Artefakten[80] entwickelt werden können.*

Diese Definition unterscheidet sich von der in der Literatur zu findenden Definition einer Produktlinie (vgl. [Sof05]) in dem Sinne, dass eine Produktlinie zwar eine Menge von Softwaresystemen ist, die aus einer gemeinsamen Menge von Artefakten entwickelt werden, zusätzlich wird diese aber durch eine Ausrichtung auf einen gemeinsamen Markt bestimmt. Diese letzte Forderung, die auf die wirtschaftliche Verwertung der Softwaresysteme abzielt, spielt im Rahmen dieser Arbeit jedoch keine Rolle. Eine Systemfamilie kann die Basis für eine oder mehrere verschiedene Produktlinien bilden[81].

Für eine Systemfamilie können Softwareanwendungen realisiert werden, die es erlauben, auf der Basis einer abstrakten Beschreibung – ggf. auch unter Nutzung von Implementationskomponenten – ein spezialisiertes Endprodukt dieser Familie zu erzeugen[82]. Die Überlegungen im Rahmen dieser Arbeit sollen dazu beitragen, dass die hierfür notwendigen Ausgangsbeschreibungen letztendlich durch Endnutzer erzeugt werden können.

Eine Weiterentwicklung von komponentenbasierten Anwendungen einer Systemfamilie kann durch das Ergänzen von Komponenten bzw. durch den Austausch vorhandener Komponenten realisiert werden (siehe [BJM+02]). Diese Möglichkeiten entstehen, da für eine Systemfamilie eine identische Anwendungsarchitektur vorgeben werden kann. Diese kann – wie es im Rahmen dieser Arbeit für Endnutzerprogrammieranwendungen geschehen ist – beispielsweise durch ein Framework vorgegeben werden, welches definierte Anknüpfungspunkte für den generierten Teil des Softwaresystems oder die Integration der Komponenten bietet. Zudem erleichtert die Einschränkung auf Domänen bzw. Systemfamilien die Auswahl bzw. das Bereitstellen im jeweiligen Kontext sinnvoll zu verwendender Komponenten.

Trotz der Ausrichtung der resultierenden Anwendungen auf einzelne Systemfamilien ist es sinnvoll, die zugrunde liegenden Konzepte soweit als möglich allgemein zu entwickeln, so dass Anpassungen auf andere Systemfamilien bzw. Domänen mit vertretbarem Aufwand möglich sind. Dieses Ziel wird auch bei dem im Rahmen dieser Arbeit exemplarisch implementierten Programmiersystem verfolgt. Die hierfür erarbeiteten Techniken und Konzepte sind nicht auf eine spezielle Domäne respektive eine Anwendungsfamilie beschränkt, selbst wenn die prototypische Implementation eine solche Ausrichtung besitzt.

Bei der Entwicklung konkreter generativer Softwaresysteme – unabhängig davon, ob diese im Bereich der Endnutzerprogrammierung Anwendung finden sollen oder nicht – müssen im Gegensatz zur „traditionellen“ Softwareentwicklung Familien von Softwaresystemen und keine Einzelsysteme im Mittelpunkt des Interesses stehen. Daraus resultieren veränderte Anforderungen an den Entwicklungsprozess und die verwendeten Analyse- und Entwurfsmethoden. Das Ziel des Entwicklungsprozesses muss die Modellierung von Gemeinsamkeiten und ggf. Unterschieden einer Menge von Softwaresystemen sein und nicht die detaillierte Beschreibung einer einzelnen Anwendung. Grundlage eines solchen Entwurfes einer Systemfamilie ist ein Modell der Zieldomäne. In der Zwischenzeit sind einige Methoden zur Analyse und Modellierung von Domänen in der Literatur dokumentiert, für einen

[80] Ein Artefakt ist in diesem Zusammenhang ein End- oder Zwischenergebnis in einem Softwareentwicklungsprozess. Dabei kann es sich sowohl um Dokumente (zum Beispiel eine Anforderungsdefinition), als auch um Programmteile (beispielsweise ein Framework oder eine Menge von Komponenten) handeln.

[81] [Whi96] fordert für eine Systemfamilie neben der Ausrichtung auf einen gemeinsamen Markt nur eine gemeinsame Menge von Eigenschaften, dadurch ist eine Produktlinie nicht mehr per se eine Systemfamilie; die Gruppierung der Softwaresysteme erfolgt dann primär durch Marketinggesichtspunkte.

[82] Dies ist auch die zentrale Idee der **generativen Programmierung** [CE00] (siehe auch Abschnitt A.4).

Überblick und eine genauere Diskussion des Verhältnisses von domänenorientierten Analyse- und Entwurfsmethoden zu denen der Objektorientierung siehe [CE00]. Die etablierten Analyse- und Entwurfsmethoden zielen derzeit noch auf die Entwicklung *„dieses speziellen Systems für diesen speziellen Kunden und Kontext“* [CE00] ab. Domänenenanalyse verschiebt den Schwerpunkt von der Codewiederverwendung hin zur Wiederverwendung von Analyse- und Designmodellen als Basis einer Familie von Softwaresystemen.

Die Beschränkung auf eine Domäne bzw. Systemfamilie führt unter anderem dazu, dass sich die Entwicklung innerhalb einer klar umrissenen Menge von Abstraktionen und Begriffen bewegt. Diese können genutzt werden, um die zu realisierenden Softwaresysteme zu beschreiben. Der folgende Abschnitt widmet sich diesem Punkt.

A.3 Domänenspezifische Sprachen

Domänenspezifische Sprachen (Domain Specific Languages, DSL[83]) sind Sprachen, die speziell für den Einsatz innerhalb eines abgegrenzten Problembereiches entworfen sind. Es kann sich dabei um eine vollwertige Programmiersprache oder um eine ausführbare Beschreibungssprache handeln, als Notation kommt neben der textuellen Form auch eine grafische Notation in Frage. In der Regel sind DSL auf den Einsatz innerhalb dieses Problembereiches begrenzt (siehe z. B. [DKV00], [CE00], [Con04]).

Entscheidend ist die Ausrichtung der Sprache auf die Anwendung innerhalb einer speziellen Domäne. Dadurch wird es möglich, die Notation und die in der Sprache verwendeten Abstraktionen so zu wählen, dass sie innerhalb der Domäne eine Bedeutung haben. Eine DSL hat den Zweck, ein Problem auf dem Abstraktionslevel der Domäne – idealer Weise sogar mit dem dort üblichen Vokabular – zu beschreiben.

Dies macht das Konzept im Hinblick auf die Endnutzerprogrammierung interessant, da dies in der Folge bedeutet, dass Domänenexperten – auch wenn sie keine Softwareentwickler sind – in der Lage sind, die in einer DSL beschriebenen Lösungen zu verstehen, zu validieren, gegebenenfalls zu ändern und eventuell sogar eigene Lösungen in einer geeigneten DSL entwickeln können. Dadurch kann eines der grundlegendsten Probleme im Softwareentwicklungsprozess adressiert werden. Software wird in der Regel durch Entwickler entworfen, die kein oder nur beschränktes Wissen im zukünftigen Anwendungsbereich der Software besitzen. Dies kann zu Missverständnissen und in der Folge zu einem nichtoptimalen Produkt führen. Durch unterschiedlichen Sprachgebrauch (Fachsprachen) sowohl auf Seiten der Domänenexperten, welche den Entwicklern die Anforderungen beschreiben, als auch auf Seiten der Softwareentwickler, die ihren Lösungsansatz beschreiben und auf eine Validierung seitens der Domänenexperten angewiesen sind, gestaltet sich der Entwurfs- bzw. Analyseprozess schwierig. Durch die Verwendung eines domänenspezifischen Vokabulars in der Programmiersprache können Entwickler und zukünftige Anwender auf einem Abstraktionslevel kommunizieren. Außerdem können DSL-Programme durch die Verwendung domänenspezifischer Abstraktionen zu einem großen Maß selbstdokumentierend gestaltet werden.

Der Begriff domänenspezifische Sprache in der neueren Literatur entspricht der Idee, die unter der Bezeichnung **Little Language** für problemspezifische Sprachen schon bei [Ben86] zu finden ist. Der Begriff „Little“ wird vom Autor unscharf gelassen, die intuitiv verständliche Idee einer Abgrenzung zu den gängigen Programmiersprachen[84] über den Sprachumfang oder die Mächtigkeit lässt sich schwer formal fassen. Beispielsweise ist SQL als Abfragesprache für relationale Datenbanken klar auf einen

[83] Im Folgenden wird die der englischen Sprache entsprechende Abkürzung DSL für domänenspezifische Sprachen verwendet.

bestimmten Anwendungsbereich eingegrenzt. Aber es wird dem Sprachumfang in keiner Weise gerecht, diese Sprache als „klein“ zu bezeichnen. Letztendlich bleibt als Charakteristikum domänenspezifischer Sprachen nur die Spezialisierung der Sprache auf einen bestimmten Problembereich, diese Eigenschaft findet sich ebenfalls bei Bentley.

In diesem Zusammenhang ist zu sagen, dass domänenspezifische Sprachen bereits seit einigen Jahren entworfen und verwendet werden, entsprechend groß ist die Anzahl konkreter Sprachen. Die systematische Beschäftigung mit diesem Forschungsbereich setzte jedoch ([DKV00] *„[…] the systematic study […] has only started more recently“*) erst deutlich später ein.

Domänenspezifische Sprachen sind nicht notwendigerweise Turing-vollständig, selbst wenn sie es sind, so sind sie in der Regel außerhalb der Anwendungsdomäne nur von geringem Nutzen. Sie lassen sich konzeptuell unter verschiedenen Gesichtspunkten betrachten, dies wird im folgenden Abschnitt näher ausgeführt.

Konzeptuelle Einordnung domänenspezifischer Sprachen

Konzeptuell kann eine domänenspezifische Sprache zunächst einfach als eine auf ein bestimmtes Problemfeld ausgerichtete Programmiersprache angesehen werden. Für gewöhnlich sind diese Sprachen dann – wie oben bereits angesprochen – vom Umfang der unterstützten Konstrukte und Schlüsselworte her im Vergleich mit GPL klein.

Eine andere Sicht auf eine domänenspezifische Sprache ergibt sich, wenn sie nicht als Programmier- sondern als Spezifikationssprache betrachtet wird. Das Vokabular domänenspezifischer Sprachen besteht in der Regel aus Konzepten einer höheren Abstraktionsebene. Dementsprechend bleiben konkrete Implementationsdetails, wie sie bei einer in einer GPL verfassten Realisierung vorkommen, verborgen. Außerdem sind die Sprachen häufig deklarativer [DKV00], also beschreibender, Natur. Dieses Betrachtungsweise rückt die DSLs in die Nähe einer Modellierungsmethode (siehe [Tol01] für domänenspezifischen Modellierungsmethodiken).

Betrachtet man domänenspezifische Sprachen stärker im Zusammenhang der Softwarearchitektur [LAB03] [CM98] so rücken wiederum andere Aspekte in den Vordergrund. Unter diesem Blickwinkel stellt eine domänenspezifische Sprache einerseits einen Parametrisierungsmechanismus dar. Je allgemeiner eine Funktion gestaltet wird, desto komplexer werden in der Folge ihre Parameter. Als Beispiel mag der Unix-Befehl Grep dienen, der zur Suche nach Zeichenketten in Dateien verwendet wird. Er akzeptiert als Eingabeparameter reguläre Ausdrücke, dabei handelt es sich um eine domänenspezifische Sprache zur Beschreibung von Zeichenketten. Ein DSL-Programm ist dann ein komplexer Parameter für eine parametrisierte Funktion (oder eine Komponente, ein Programm). Andererseits kann eine DSL als ein domänenspezifisches Interface zu einer Funktionsbibliothek gesehen werden. Durch die Verwendung von Funktionen der Bibliothek können Sachverhalte auf einer abstrakteren Ebene formuliert werden, als dies mit der zugrunde liegenden Programmiersprache möglich ist.

Domänenspezifische Sprachen können auch als Benutzerschnittstelle verwendet werden; [Ben86] beschreibt beispielsweise die Sprache PIC[85] unter diesem Gesichtspunkt. Der oben angesprochenen Grep-Befehl ist ein weiteres Beispiel für die Verwendung einer DSL als Benutzerschnittstelle. Bei der Ausrichtung auf einen Endnutzer als Anwender der Sprache ergeben sich nochmals höhere

[84] In der englischsprachigen Literatur ist i. A. die Rede von General Purpose Languages, also Mehrzwecksprachen (C, C++, Java, Delphi, etc.). Im Folgenden wird auch in diesem Text die Abkürzung GPL für diese Gruppe von Programmiersprachen verwendet.

[85] Die Sprache PIC wurde von Brain W. Kernighan entworfen und dient zur Beschreibung von Grafiken (zu PIC siehe [Ker91]).

Anforderungen an die Nutzbarkeit und Verständlichkeit der Sprache im Vergleich zu Szenarien, bei denen der Anwender der DSL ein Softwareentwickler ist.

Betrachtet man die Einordnung einer domänenspezifischen Sprache in den Softwareentwurfsprozess, so kann sie als ein Konzept zur Wiederverwendung gesehen werden. Wenn die DSL bereits entworfen und implementiert ist, besteht ein – wie auch immer geartetes – Umsetzungskonzept für die durch die Sprache bereitgestellten Abstraktionen. Die Verwendung der DSL entspricht dann der Beschreibung eines Softwaresystems mittels vorhandener Bausteine. Ein solches Vorgehen wird im Software-engineering allgemein als Bottom-Up-Vorgehen[86] bezeichnet.

Umgekehrt kann man die domänenspezifische Sprache wie oben bereits angesprochen als Spezifikationssprache sehen. Das Vokabular und die Regeln dieser Sprache können rein aus dem Anwendungsbereich heraus motiviert sein und festgelegt werden, ohne dass eine konkrete Umsetzung bekannt ist. Das Softwaresystem wird in diesem Fall auf einer abstrakten Ebene beschrieben, ohne dass Implementationsdetails festgelegt sind. Diese werden in weiteren Schritten ergänzt. Ein solches Vorgehen entspricht einem Top-Down-Ansatz.

Grundsätzlich können domänenspezifische Sprachen sowohl imperativer als auch deklarativer Natur sein. Die in der Literatur vorhandenen Beispiele sind häufig deklarativ; in [LR94] findet sich eine sehr interessante Gegenüberstellung zu diesem Punkt. In diesem Artikel ist die Entwicklung und der Einsatz einer domänenspezifischen Sprache für die Software von Telefonswitches bei den AT&T Bell Laboratories beschrieben. Zunächst wurde eine imperative, Turing-vollständige Sprache entworfen (PRL); diese wurde später durch eine deklarative Variante (PRL5) ersetzt. Beim Vergleich der beiden Sprachen kommt der Autor zu dem Schluss, dass die deklarative Variante im betrachteten Projekt eine wesentlich weiter gehende Automatisierung erlaubte und zu reduzierten Kosten und verbesserter Qualität der resultierenden Implementationen führte. Begründet wird dies dadurch, dass sich aus der deklarativen Beschreibung durch unterschiedliche Verarbeitungen bzw. Prozessoren alle notwendigen Artefakte erzeugen lassen. Die imperative Spezifikation legte durch die Beschreibung des notwendigen Vorgehens eine Interpretation[87] fest. Bezogen auf die vorherigen Ausführungen bedeutet dies, dass eine deklarative DSL zur Verwendung im Rahmen eines Top-Down-Prozesses besser als eine imperative Variante geeignet ist.

Eine domänenspezifische Sprache wird teilweise auch als Beschreibung einer Programmfamilie bzw. als Mechanismus zur Festlegung eines konkreten Exemplars einer Programmfamilie gesehen [CM98]. Dies ist eine zutreffende Sicht für eine bestimmte Gruppe domänenspezifischer Sprachen. Wenn auf Basis einer Beschreibung in einer DSL durch einen Generator (vgl. Abschnitt A.5) eine Softwareanwendung erzeugt wird, dann legt diese Beschreibung ein konkretes Exemplar einer Programmfamilie fest. Im Falle der bereits angesprochenen domänenspezifischen Sprachen SQL, PIC und der regulären Ausdrücke trifft diese Sichtweise jedoch nicht zu. Dieser Punkt zeigt noch einmal deutlich, wie schwierig es ist, die unter dem Begriff domänenspezifische Sprachen existierende Vielzahl an konkreten Beispielen und Konzepten unter allgemeingültigen Gesichtspunkten zu beschreiben. Die eben dargelegten konzeptuellen Betrachtungen und die im Folgenden beschriebenen unterschiedlichen Realisierungsansätze bieten jedoch zumindest ein gewisses Raster zur Einordnung domänenspezifischer Sprachen.

[86] Für Bottom-Up- und Top-Down-Entwurf siehe z. B. [PB96].

[87] Ziel des Projektes war die Beschreibung von Integritätsbedingungen auf einer Datenbank. In PRL lies sich beschreiben, wie diese Bedingungen zu prüfen waren, PRL5 erlaubte die Beschreibung der Bedingungen selbst.

Realisierung Domänenspezifischer Sprachen

Da es sich auch bei der im Rahmen des prototypisch implementierten Programmiermodells verwendeten grafischen Notation um eine domänenspezifische Sprache handelt, werden in diesem Abschnitt einige der gängigen Ansätze zur Realisierung solcher Sprachen vorgestellt.

Eine domänenspezifische Sprache kann dadurch verwirklicht werden, dass ein geeigneter Compiler bzw. Interpreter entworfen und implementiert wird, dies entspricht dem „klassischen" Weg, eine neue Programmiersprache zu entwerfen. Als Vorteil für dieses Vorgehen ist die Verfügbarkeit von Werkzeugen zur Erzeugung von Compilern zu nennen (z. B. Yacc [Lex05], Coco/R [MWL05]). Außerdem erlaubt es dieser Ansatz, die DSL ohne – bei anderen Ansätzen lässt sich dies zum Teil nicht vermeiden – Zugeständnisse an die Notation zu entwerfen. Ein weiterer Vorteil ist die Möglichkeit, Fehlerbehandlungen, statische Analysen und Optimierungen auf der Ebene der Domänenabstraktionen durchführen zu können ([DKV00], hier finden sich auch Verweise auf Werkzeuge, die diesen Ansatz unterstützen). Da keine Zwischenschritte oder Vorverarbeitungen erforderlich sind, ist dieser Ansatz in der Regel sehr effizient. Nachteilig ist der vergleichsweise hohe Aufwand und eventuell die Notwendigkeit entsprechender Kenntnisse im Compilerbau für den Entwurf und auch die Wartung der DSL. Die Gruppe dieser DSL wird in der Literatur als **freistehende** DSL bezeichnet [CE00], Beispiele für diese Gruppe sind SQL und T_EX.

Alternativ kann eine DSL durch die Erweiterung einer bereits vorhandenen Basissprache realisiert werden (**eingebettete** domänenspezifische Sprache, domain specific embedded language, DSEL). Dies hat den Vorteil, dass die in der Basissprache bereits vorhandenen Sprachkonstrukte genutzt werden können und nicht neu implementiert werden müssen. Dieser Ansatz lässt sich – je nachdem in welcher Form die Basissprache verwendet wird – weiter unterteilen (vgl. [DKV00], eine detailliertere Unterscheidung ist in [CE00] zu finden):

1. ***Eingebettete Sprachen oder domänenspezifische Bibliotheken***[88]

   Bei diesem Ansatz werden die Kapselungs- und Abstraktionsmechanismen der Basissprache verwendet um die Idiome der Domäne auszudrücken. Dies geschieht beispielsweise durch die Definition von Klassen, Methoden oder Operatorüberladung. Je nach Basissprache können auch Techniken aus der Metaprogrammierung[89] angewandt werden, ein bekanntes Beispiel hierfür ist die Nutzung der Templatefunktionalität in C++ zur Realisierung domänenspezifischer Sprachen. Vorteilhaft ist die Tatsache, dass der Compiler bzw. Interpreter der Basissprache ohne Modifikationen verwendet werden kann und in der Regel auch eine entsprechende Unterstützung durch Softwarewerkzeuge existiert. Dieser Ansatz ist jedoch durch die von der Basissprache bereitgestellten Konzepte beschränkt. Häufig muss bei diesem Vorgehen daher auf die aus Domänensicht optimale Notation aufgrund der syntaktischen Beschränkungen der Basissprache verzichtet werden, als Beispiel mag die Realisierung einer DSL für Matrizenrechnungen in C# dienen, die mathematische Notation könnte hierbei nicht beibehalten werden. Nachteilig ist außerdem die zu große Mächtigkeit der resultierenden domänenspezifischen Sprache, da alle Konzepte aus der Basissprache hier ebenfalls verfügbar sind. Dies ist insbesondere für die Anwendung im Bereich der Endnutzerprogrammierung von Nachteil.

[88] Die Abgrenzung zwischen einer Bibliothek im herkömmlichen Sinne und einer DSL ist schwer zu formalisieren, beide fügen der Basissprache neue Abstraktionen hinzu und erweitern das Vokabular.

[89] Metaprogrammierung bezeichnet das Erstellen von Programmen, welche andere Programme oder sich selbst manipulieren.

2. ***Präprozessoren oder Makroprozessoren***

   Hier werden die Anweisungen der DSL durch einen Vorverarbeitungsschritt in Ausdrücke der Basissprache übersetzt. Dieser Ansatz zeichnet sich durch seine Einfachheit hinsichtlich der Umsetzung aus. Nachteilig ist, dass Optimierungen und statische Typprüfungen nicht auf dem Level der DSL realisierbar sind, Fehlermeldungen können dem Nutzer der DSL in der Regel nur auf dem Level der Basissprache mitgeteilt werden.

3. ***Erweiterbare Compiler bzw. Interpreter***

   Dieser Ansatz ist dem unter 2 aufgeführten ähnlich. Allerdings findet hier eine Einbettung des Vorverarbeitungsschrittes in den Compiler der Basissprache statt, dies erlaubt in der Regel eine bessere Typprüfung bzw. Optimierungen. Ein solches Vorgehen ist technisch recht anspruchsvoll.

Neben den eben genannten Möglichkeiten, eine vorhandene Basissprache zu erweitern, kann eine Programmiersprache auch wiederverwendet werden, indem sie als Ausgabeformat eines DSL-Prozessors verwendet wird. Die so erzeugte Repräsentation des Programms kann dann mittels eines Compilers oder Interpreters für die Basissprache übersetzt bzw. ausgeführt werden. Der Vorteil dieses Ansatzes liegt darin, dass bei der Realisierung der DSL auf die Möglichkeiten der Basissprache zurückgegriffen werden kann, um die in der DSL bereitgestellten Abstraktionen zu realisieren. Der Unterschied zu dem unter 2 angeführten Ansatz, Prä- bzw. Makroprozessoren zu verwenden, liegt darin, dass die domänenspezifische Sprache hierbei nicht innerhalb der Basissprache realisiert wird. Dies erleichtert es, Features der Basissprache auszublenden. Außerdem kann (je nach Mächtigkeit des DSL-Prozessors) eine von der Basissprache unabhängige und den Erfordernissen der Domäne angepasste Notation gewählt werden. Dieser Ansatz teilt jedoch die übrigen oben genannten Nachteile.

Bentley führt in [Ben86] aus, wie durch den Einsatz einer so genannten **Zwischensprache** (intermediate language) der Aufwand der Portierung von domänenspezifischen Sprachen auf unterschiedliche Plattformen reduziert werden kann. Statt eine Menge unterschiedlicher domänenspezifische Sprachen (Anzahl sei M) jeweils für alle Zielplattformen (Anzahl sei N) zu übersetzen, wird jede dieser Sprachen in dieselbe Zwischensprache übersetzt. Für diese muss dann für jede Zielplattform ein Compiler existieren. Damit werden statt M*N nur M+N Compiler benötigt. Zusammen mit dem oben angesprochenen Vorgehen, eine vorhandene Basissprache als Ausgabe eines DSL-Prozessors zu verwenden, ergibt sich ein interessanter Weg für den Entwurf einer domänenspezifischen Sprache. Wählt man als Ausgabe des DSL-Prozessors eine auf einer möglichst großen Menge von Betriebssystemen verfügbare GPL, so ist die DSL ohne zusätzlichen Aufwand auf derselben Menge von Plattformen verfügbar. Dieses Vorgehen entspricht prinzipiell der Verwendung von P-Code bzw. der Verwendung virtueller Maschinen [Ter05].

In [CE00] werden modulare, komponierbare DSLs als weitere Realisierungsmöglichkeit angesprochen. Die Komposition von DSLs im Sinne einer Hintereinanderausführung ihrer Prozessoren findet sich schon bei Bentley [Ben86]. Czarnecki spricht aber auch einen über diese einfache Komposition hinausgehenden Ansatz an, bei dem eine Sprache die Semantik der Sprachen beeinflusst, mit denen sie zusammengesetzt wird. Dieser Ansatz wird als Aspekt-DSL bezeichnet, analog zum aspektorientierten Programmieren, da sie an mehreren Stellen Änderungen bewirken und diese die modularen Strukturen der Sprachen kreuzen. Als Beispiel wird eine Sprache angeführt, die es erlaubt, Fehlerbehandlung in andere Sprachen einzuführen. Eine solche Sprachinteraktion setzt eine gemeinsame Infrastruktur der beteiligten DSL voraus.

In Abschnitt A.5 Generatoren wird die Realisierung domänenspezifischer Sprachen durch Generatoren behandelt, also eigenständige Anwendungen, welche eine domänenspezifische Sprache in der Regel auf Abstraktionen einer Vielzweckprogrammiersprache abbilden. Zunächst sollen jedoch einige der Vor- und Nachteile diskutiert werden, die bei einer Nutzung domänenspezifischer Sprachen zu erwarten sind.

Vor- und Nachteile dömänenspezifischer Sprachen

Wie oben bereits dargelegt ist das Spektrum der domänenspezifischen Sprachen sehr breit. Einige der im Nachstehenden diskutierten Vor- bzw. Nachteile sind in der Folge nur für Teilmengen der domänenspezifischen Sprachen relevant. Dies wird an den entsprechenden Stellen kenntlich gemacht. Der Schwerpunkt liegt auf der Verwendung domänenspezifischer Sprachen zum Entwurf von Softwaresystemen, da diese Verwendung im Zusammenhang mit der vorliegenden Arbeit relevant ist.

Ein bereits mehrfach genannter Vorteil beim Einsatz domänenspezifischer Sprachen zur Entwicklung von Anwendungen ist die Erhöhung des Abstraktionsniveaus auf eine auch von Domänenexperten verständliche Ebene. Dies hat zur Folge, dass die Spezifikation auch von diesen verstanden und validiert werden kann, sie wird selbstdokumentierend. Im Idealfall kann eine Anwendung von einem Nichtsoftwareentwickler mit ausreichendem Wissen im Anwendungsbereich erstellt werden. Dadurch können Fehler beim sonst notwendigen Wissenstransfer von den Domänenexperten hin zu den Entwicklern der Anwendung vermieden werden. Hier ist jedoch – und das wird in der Regel in der „Pro-DSL-Literatur“ nicht diskutiert – zu beachten, dass es sich im Wesentlichen um eine Verlagerung des ursprünglichen Problems handelt. Eine Spezifikation in einer domänenspezifischen Sprache muss durch einen geeigneten Prozessor in die Zielanwendung oder eine Vorstufe übersetzt werden. Die Implementation dieses Werkzeuges setzt Kenntnisse aller relevanten Konzepte der zu modellierenden Domäne und ihrer Interaktionen inklusive eventueller Randbedingungen voraus. Hier ist in der Regel ein weitaus größeres Domänenwissen vonnöten, als bei der Erstellung einer einzelnen Anwendung nach einem herkömmlichen Ansatz.

Auch der Vorteil der Validierung durch den Domänenexperten muss relativiert werden. Er kann die Korrektheit der Spezifikation in der domänenspezifischen Sprache prüfen. Die Korrektheit des Prozessors, der die Übersetzung vornimmt, kann nur in Zusammenarbeit mit Softwareentwicklern geprüft werden.

Ein Nachteil des Einsatzes domänenspezifischer Sprachen für die Softwareentwicklung ist bereits angeklungen. Es entsteht ein vergleichsweise hoher Anfangsaufwand für den Entwurf der Sprache und der notwendigen Werkzeuge. Hinzu kommen die Erstellung von Dokumentation und Schulungen für die zukünftigen Anwender der Sprache. Insbesondere wenn die Sprache nicht nur innerhalb eines festen Teams über mehrere Projekt hinweg eingesetzt werden soll, kommt diesem Punkt hohe Bedeutung zu. Dieser Aufwand ist nur dann gerechtfertigt, wenn die DSL in mehreren Projekten zum Einsatz kommen kann. In diesem Fall ist eine domänenspezifische Sprache eine Möglichkeit zur Wiederverwendung von Wissen um die Realisierung der Konzepte eines Anwendungsbereiches über verschiedene Applikationen hinweg.

Abhängig von der Form der Realisierung schafft der Einsatz einer DSL zur Beschreibung einer Anwendung eine zusätzliche Abstraktionsebene. Dies hat den Vorteil, dass Änderungen an der Implementation eines Systems, solange sie nicht die Fachlogik bzw. die aus der Anwendungsdomäne heraus motivierte Lösungsbeschreibung betreffen, durch Veränderungen am DSL-Prozessor erfolgen können. Idealerweise erhält man eine strikte Trennung von Fachlogik und Implementationsdetails. Die Beschreibung der Fachlogik als solche ist unabhängig von einer konkreten Realisierung.

Darüber hinaus können aus einer einzelnen Beschreibung der Anwendungslogik unterschiedliche, angepasste Realisierungen – beispielsweise für unterschiedliche Hardwareplattformen – durch den Einsatz verschiedener Prozessoren generiert werden.

In [CE00] wird auf die Gefahr hingewiesen, dass durch den Einsatz domänenspezifischer Sprachen, insbesondere freistehender DSLs, unter Umständen Insellösungen entstehen, die sich nur schlecht mit anderen Systemen integrieren lassen. Deursen stellt in [DK98] die Frage, wie Anwendungen, die in unterschiedlichen domänenspezifischen Sprachen realisiert sind, zusammenarbeiten können. Nach meiner Ansicht muss die spätere Integration mit anderen Anwendungen wie beim Entwurf einer normalen Applikation berücksichtigt werden, stellt jedoch kein DSL-spezifisches Problem dar. Die projektübergreifende Wiederverwendung von Quelltextteilen ist jedoch auf Projekte beschränkt, die entweder dieselbe DSL verwenden, oder aber in denen eine Programmiersprache Anwendung findet, in welche die DSL übersetzt werden kann.

Weitere, weniger nahe liegende Vorteile ergeben sich aus der Anwendung domänenspezifischer Sprachen zur Spezifikation. In der Regel werden Anforderungen von Domänenanwendern in natürlicher Sprache beschrieben. Natürliche Sprache ist in ihrer Bedeutung nicht eindeutig festgelegt, daher resultiert dies in mehrdeutigen Spezifikationen. Eine DSL kann formal beschrieben werden und verfügt über ein Vokabular mit festgelegten Bedeutungen, in der Folge ist eine von einem Domänenexperten in einer DSL verfasste Spezifikation eindeutig. Darüber hinaus existiert in Form des DSL-Prozessors eine klare Abbildungsvorschrift, wie eine Spezifikation in die zugehörige Implementation umgesetzt wird, dies ist bei Spezifikationen in natürlicher Sprache nicht der Fall. Da diese Abbildungsvorschrift außerdem automatisch ausgeführt werden kann, befinden sich Implementation und Spezifikation immer im Einklang, es kommt nicht zu der Situation, dass mühsam erarbeitetes Wissen über einen Anwendungsbereich oder eine Applikation nur noch im Quellcode niedergelegt ist.

A.4 Generatives Programmieren und Codegenerierung

Nicht zuletzt aufgrund der steigenden Anforderungen an die Entwicklung von Software in den Bereichen Qualität, Produktivität der Entwickler und Entwicklungskosten entstand die Überlegung, Teile der Entwicklung zu automatisieren. In diesem Zusammenhang fallen häufig die Begriffe Codegenerierung bzw. Generatives Programmieren; an vielen Stellen werden diese auch synonym gebraucht, dies ist jedoch nicht korrekt.

Generatives Programmieren ist ein Paradigma in der Softwareentwicklung mit dem Ziel, aus einer abstrakten Beschreibung eines Softwaresystems weitgehend automatisiert eine konkrete Realisierung dieser Software zu erzeugen. Eine der zentralen Arbeiten zum Generativen Programmieren ist [CE00], hier wird das Generative Programmieren als eine direkte Konsequenz der so genannten **Automation Assumption** [CE99] gesehen:

> *„If you can compose components manually, you can also automate this process“*

Nach der in dieser Arbeit angegebenen Definition basiert das Generative Programmieren auf der Modellierung von Systemfamilien (siehe A.2 Domänen und Systemfamilien) und hat zum Ziel, aus einer gegebenen Anforderungsdefinition automatisiert ein angepasstes End- oder Zwischenprodukt auf der Basis elementarer Implementationskomponenten zu erstellen.

Zum einen werden bei dieser Definition Komponenten zwangsläufig als Basis des Erstellungsprozesses angesehen. Zum anderen ist nur die Rede von automatisierter Erstellung; dies kann aber

auch durch das Einsetzen von spezialisierten Komponenten in einen entsprechenden Anwendungsrahmen geschehen.

Codegenerierung im Sinne des Erzeugens einer textuellen Repräsentation eines Programms, wie es beispielsweise beim automatisierten Erzeugen von Datenobjekten zu vorhandenen Datenbanktabellen vorkommt [Dol04], ist nach dieser Definition nicht zwangsläufig Bestandteil des Generativen Programmierens. In der praktischen Realisierung der zugehörigen Konzepte ist es jedoch eine häufig verwendete Technik.

Im Unterschied zum sog. Automatischen Programmieren, einer verwandten Idee aus dem Bereich der KI (siehe z. B. [Bal85]) zielt das Generative Programmieren nicht zwangsläufig auf den höchstmöglichen Level der Automatisierung ab, vielmehr sind auch die Möglichkeiten Teilschritte zu automatisieren Gegenstand der Untersuchungen.

Zweifelsohne hat die komponentenbasierte Wiederverwendung bei der automatisierten Erstellung von Anwendungen große Vorteile. Es gibt jedoch eine Reihe von Anwendungen, bei denen ganze Programme oder Programmteile ohne Verwendung von Komponenten durch den Einsatz spezieller Softwarewerkzeuge, so genannter Generatoren, erzeugt werden. Diese Werkzeuge und die Forschungen in diesem Bereich gehören nicht notwendigerweise in den Bereich des Generativen Programmierens. Da die zugrunde liegenden Konzepte im Rahmen dieser Arbeit eine Rolle spielen, werden sie im Folgenden näher betrachtet.

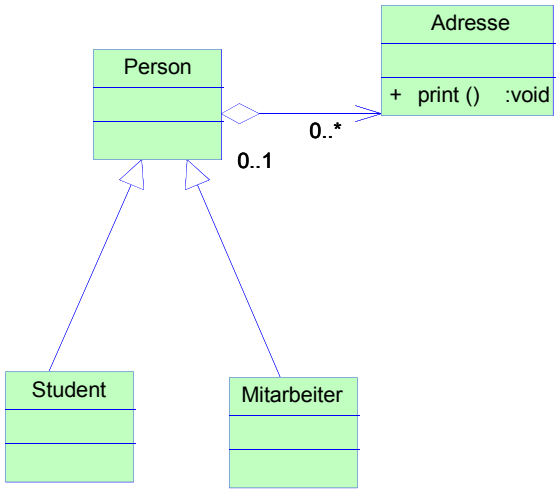
A.5 Generatoren

Einleitung

Überlegungen zu Generatoren finden sich schon seit einigen Jahren immer wieder in der informatischen Forschungsliteratur. Vom Standpunkt der angewandten Informatik gewinnt diese Technik in den letzten Jahren zunehmend an Bedeutung, oder wie es in [Dol04] formuliert ist: *„Generating Code isn't new. What's new is that it actually works in the real world."*[90].

Generatoren existieren beispielsweise in CASE-Tools (Computer Aided Software Engineering) zur Erzeugung von Quelltexten in einer Zielprogrammiersprache aus UML-Modellen. Diese Generatoren sind aber in der Regel auf eine direkte Übersetzung der in der UML vorhandenen Konzepte in die Zielsprache beschränkt und realisieren im Wesentlichen Standardimplementationen von Klassen-, Attribut- und Methodenschablonen. Dem Entwickler wird Arbeit abgenommen, er wird von Routineaufgaben entlastet, der resultierende Quelltext ist jedoch noch keine Realisierung des Zielsystems.

[90] „Code generieren ist nicht neu. Neu ist, dass es tatsächlich in der wirklichen Welt funktioniert."




```
using System;

public class Adresse
{
    public void Print()
    {
        // TODO: implement
    }
}

public class Person
{
    public Adresse[] Association1;
}

public class Mitarbeiter : Person
{}
public class Student : Person
{}
```

Abb. A.2: UML-Diagramm und zugehöriger generierter Quellcode

In Abb. A.2 ist ein einfaches UML-Modell und der daraus mit einem kommerziellen CASE-Tool[91] erzeugte Quelltext in C# zu sehen. Gut zu erkennen ist hier, dass der Generator lediglich die in UML dargestellten Konzepte in eine andere Darstellung überführt hat, es sind keine[92] wesentlichen neuen Details in der Implementation hinzugekommen. Dies ist nachvollziehbar, wenn man sich vor Augen führt, dass je nach Anwendungsbereich des obigen Modells beispielsweise die Methode `Print()` der Klasse Adresse ein Versandetikett auf einem Drucker, einen Überblick auf dem Bildschirm oder anderes ausgeben kann. Diese Entwurfsdetails sind aus dem Ausgangsmodell jedoch nicht abzuleiten.

Im Bereich der Unified Modeling Language [Obj03b] existiert mit der Model-Driven Architecture (MDA) Initiative der OMG eine interessante Entwicklung zur Generierung von Softwaresystemen aus plattformunabhängigen Modellen. Diese Technologie war in den letzten Jahren Gegenstand umfangreicher Forschungen und Veröffentlichungen. Zur Zeit ist es jedoch noch nicht möglich, beliebige Softwaresysteme auf diese Art und Weise zu entwickeln (siehe [HK04]).

Zum Zeitpunkt dieser Arbeit existiert nach meinem Kenntnisstand noch kein allgemeiner Programmgenerator, der ohne Einschränkungen auf der Basis einer abstrakten Anforderungsdefinition eine Realisierung eines beliebigen gewünschten Softwaresystems erzeugt. Allgemeine Programmgeneratoren setzen eine ausreichend implementationsnahe Beschreibung des zu erzeugenden Softwaresystems voraus, die verwendeten Modellierungssprachen sind dementsprechend komplex und von Nichtprogrammierern nicht zu handhaben. Für den Bereich der Endnutzerprogrammierung sind solche Werkzeuge daher nicht geeignet. Um in diesem Bereich nutzbare Ergebnisse erzielen zu können, sind Einschränkungen der Anwendungsbereiche der realisierten Werkzeuge auf Domänen bzw. Systemfamilien sinnvoll.

[91] Es handelt sich hier um das Produkt PowerDesigner, Version 10 der Firma Sybase [Syb06]

[92] Die Realisierung der Aggregationsbeziehung durch ein öffentliches Array ist der einfachste Ansatz zum Abbilden einer solchen Beziehung. Da bei einer Aggregationsbeziehung jedoch das Ganze stellvertretend für seine Teile handeln soll und kein direkter Zugriff auf diese von Außerhalb erfolgen soll, wäre die Verwendung eines privaten Arrays und entsprechender Zugriffsmethoden der bessere Ansatz zur Realisierung.

Definition

Um aus einer abstrakten Beschreibung ein konkretes, lauffähiges Softwaresystem zu erzeugen ist zum einen eine Spezifikationsnotation, dies kann beispielsweise eine domänenspezifischen Sprache sein, notwendig. In dieser wird das Zielsystem auf einer abstrakten Ebene beschrieben. Zum anderen muss eine Abbildung, die diese abstrakte Beschreibung in eine ausführbare Softwarelösung überführt, existieren. Im Folgenden wird eine Möglichkeit zur Realisierung dieser Abbildung betrachtet.

Verallgemeinernd kann man sagen, dass Codegenerierung darauf abzielt, Programme zu schreiben, die Programme schreiben. Ein Generator kann wie folgt definiert werden:

> *Ein Generator ist eine Anwendung[93] mit der Aufgabe eine Repräsentation bzw. ein Modell einer Softwareapplikation oder eines Teils davon in eine weniger abstrakte Repräsentation zu überführen. Als Teil einer Software werden hierbei Bestandteile der Software gesehen, die für eine vom Rest getrennte Erzeugung hinreichend gekapselt sind. Dies kann beispielsweise eine Routine, ein Modul, eine Klasse, eine Komponente oder aber auch ein komplettes Softwaresystem sein.*

Das Resultat eines einzelnen Generatorschrittes muss nicht bereits die Implementierung im Sinne einer Repräsentation auf der untersten durch einen Entwickler genutzten Abstraktionsebene des jeweiligen Modells sein. Der Prozess vom Ausgangsmodell hin zu dieser Ebene kann eventuell durch Hintereinanderausführung mehrerer Generatoren realisiert werden[94]. Der Generatorbegriff ist in dieser Arbeit sehr allgemein definiert und beinhaltet keinerlei Festlegung hinsichtlich einer konkreten technologischen oder konzeptionellen Umsetzung. Außerdem ist zu bemerken, dass die als Eingabe des Generators verwendete Repräsentation auch eine leere Eingabe sein kann. Generatoren, die keine andere Eingabe als die leere akzeptieren, erzeugen – wenn man einen determinierten Generierungsalgorithmus voraussetzt – immer ein identisches Ergebnis. Dieser Fall ist jedoch im Allgemeinen nur von eingeschränktem Nutzen und im Folgenden nicht von Interesse. Anwendungen, welche nur Transformationen auf derselben Abstraktionsebene (z. B. ein Verschieben von Methoden innerhalb des Programmcodes) oder von einer konkreteren auf eine allgemeinere Ebene – beispielsweise zum Erzeugen eines Klassendiagramms aus Programmquelltexten – durchführen, sind nach dieser Definition keine Generatoren.

Um die obige Definition zu vervollständigen, muss noch der verwendete Modellbegriff konkretisiert werden. Dieser unterscheidet sich von dem in Kapitel 4 im Zusammenhang mit Programmiermodellen verwendeten.

> *Im Zusammenhang mit der Betrachtung von Generatoren wird in dieser Arbeit ein Artefakt als Modell bezeichnet, wenn es einen Teil einer Softwareapplikation beschreibt, aber nicht direkt ausführbar ist, d. h. es muss zunächst in eine konkretere Form überführt werden (vgl. [Voe03]).*

Ein Modell im Sinne dieser Definition muss nicht notwendigerweise eine grafische Repräsentation eines Sachverhaltes darstellen, auch eine textuelle Beschreibung einer Applikation ist nach dieser Definition ein Modell.

Bezogen auf die am Anfang dieses Anhangs gegebene Definition einer Anwendungsfamilie handelt es sich bei einem Generator um einen Instanzierungsmechanismus, der auf Basis der Eingangsdaten konkrete Exemplare dieser Familie von Anwendungen erzeugt. Dieser Umstand ist unter dem

[93] Der Generator kann auch Bestandteil eines Programms sein, Anwendung ist hier nicht im Sinne einer eigenständigen Applikation gemeint.

[94] In der Literatur findet sich auch (z. B. [Cle88]) die Definition, dass ein Generator eine Eingabespezifikation in ein ausführbares Programm übersetzt, dies ist für die in dieser Arbeit betrachten Generatoren zu einschränkend.

Gesichtspunkt der Wiederverwendung interessant, da es dadurch möglich wird, nicht nur einen Algorithmus oder eine Datenstruktur sondern einen kompletten Anwendungsentwurf wieder zu verwenden. Die Verwendung von Frameworks ist ein alternatives Mittel, um dieses Ziel zu erreichen.

Zu den Hauptgründen, Techniken zur Codegenerierung einzusetzen, gehört neben der dadurch erreichbaren Flexibilität die Möglichkeit, durch diesen Mechanismus domänenspezifische Sprachen (siehe A.3 Domänenspezifische Sprachen) zu realisieren und damit Nichtprogrammierern bzw. Domänenexperten in einem beschränkten Umfang die Entwicklung eigener Anwendungen zu ermöglichen. Durch den Einsatz von Generatoren lässt sich das Abstraktionsniveau bei der Softwareentwicklung anheben. Konzeptionell ist die Codegenerierung in diesem Fall der Schritt vom abstrakten, domänenbasierten Modell hin zur Implementation. Der Generator beinhaltet in einem solchen Szenario Wissen um die Realisierung von Domänenabstraktionen auf einer niedrigeren Ebene und ist damit auch eine Möglichkeit der Wiederverwendung dieses Wissens.

An vielen Stellen findet sich in der Literatur die Definition, ein Generator sei ein Compiler für eine domänenspezifische Sprache (z. B. [BCR+00]). Diese Betrachtungsweise hat zur Folge, dass eine Vielzahl von Anwendungen mit eindeutig generativem Charakter nicht unter die Definition eines Generators fallen, da die Eingangsdaten nicht der Definition einer DSL genügen. Dies ist beispielsweise bei einem durch Interaktion in Form von Menüs gesteuerten Generator der Fall. Oder aber die Definition einer DSL muss entsprechend erweitert werden; dies würde jedoch keine sinnvolle Abgrenzung mehr ermöglichen.

Deutlich wird dieses Problem auch an datenorientierten Generatoren, die z. B. aus in einer relationalen Datenbank abgelegten Werten[95] eine Anwendung generieren. Eine solche Anwendung sollte unter die Definition „Generator“ fallen. Die Eingangsdaten für diese Anwendung als domänenspezifische Sprache zu bezeichnen ist in den meisten Fällen wenig sinnvoll. Daher wird in dieser Arbeit der Standpunkt vertreten, dass Generatoren ein Weg sind, um domänenspezifische Sprachen zu realisieren bzw. diese eine Möglichkeit darstellen, um die Eingangsdaten eines Generators zu spezifizieren. Trotz der Nähe der beiden Konzepte sollten die Definitionen jedoch unabhängig sein.

In [Sma04] wird die Unterscheidung zwischen einem Generator als Werkzeug und einem Generator als Sprache betont. Ein Generator als Sprache wird dabei als geschlossenes System gesehen, dessen Ausgabe in der Regel nicht bearbeitet werden muss. Bei der Verwendung eines Generators als Werkzeug wird mehr Verantwortung an den Nutzer delegiert, hier kann es vorkommen, dass der generierte Code verstanden und modifiziert werden muss. Folgt man dieser Betrachtungsweise, so ist klar, dass die Zielgruppe der Generatoranwendung eng mit der Unterscheidung zusammenhängt. Einem Nichtentwickler ist ein „Generatorwerkzeug“ nach dieser Definition nicht zuzumuten. Da die Begriffe Werkzeug und Sprache in diesem Umfeld schon überbesetzt sind, wird im Folgenden – wenn nötig – eine Unterscheidung nach Zielgruppen getroffen um für Klarheit im Sprachgebrauch sorgen.

Der einfachste denkbare Fall eines Generators ist die Ausgabe einer Zeichenkette per WriteLine-Direktive. Es bedarf keiner großen Überlegungen um zu sehen, dass dieser Ansatz in seiner Anwendung recht limitiert ist. Im Folgenden werden daher Konzepte und Realisierungsmöglichkeiten für Generatoren diskutiert, welche es erlauben, weitergehende Ansätze umzusetzen.

[95] Hier sind Werte im Sinne von Daten gemeint. Obwohl es prinzipiell möglich ist, in einer DSL formulierte Programmbeschreibungen in einer Datenbank abzulegen, wird dieses Szenario hier nicht betrachtet.

Klassifikation von Generatoren

Czarnecki und Eisenecker beschreiben in [CE00] eine interessante Klassifikation für Generatoren. Diese erlaubt es, Generatoren nicht nur anhand der zur Realisierung verwendeten Technologie zu klassifizieren, sondern ermöglicht eine Einteilung auf Basis der durchgeführten Transformationen im Hinblick auf die modulare Struktur der Zielanwendung. Da die durchgeführten Transformationen der zentrale Aspekt eines Generators sind [BS99], liegt es nahe, eine Einteilung auf dieser Basis durchzuführen. Diese Klassifikation wird im Folgenden auf die im Rahmen dieser Arbeit verwendete Generatordefinition bezogen.

Generatoren, die eine Transformation einer Repräsentation auf einem höheren Level in eine Repräsentation auf einem niedrigeren Level durchführen, so dass die modularen Strukturen der höheren Repräsentation erhalten bleiben, werden als **Kompositionsgeneratoren** bezeichnet. Dabei wird jedes Modul (oder Abstraktion/Repräsentation) des höheren Levels durch ein oder mehrere Module (Abstraktionen/Repräsentationen) des niedrigeren Levels implementiert. Es findet also eine hierarchische Dekomposition[96] statt. Beispiele für solche Transformationen sind die Wahl eines Algorithmus zur Lösung einer Aufgabe und die Datentypverfeinerung. Ersteres bezeichnet das Ersetzen einer deklarativen Spezifikation durch eine Methode, die diese Spezifikation erfüllt. Unter Letzterem wird die Implementation eines spezifizierten Datentyps auf der niedrigeren Ebene verstanden, z. B. eines binären Baumes durch entsprechende Zeigerstrukturen. Generatoren, die nach diesem Prinzip arbeiten, berücksichtigen die aus dem Softwareengineering bekannten Prinzipien der schrittweisen Verfeinerung und der Kapselung. Dieser Generatortyp findet häufig in Werkzeugen zum Entwurf von grafischen Benutzerschnittstellen oder CASE-Werkzeugen Verwendung.

Der Vollständigkeit halber sind auch solche Transformationen zu erwähnen, die keine Konkretisierung des Modells vornehmen, sondern nur eine Änderung auf derselben Abstraktionsebene bewirken. Hierzu zählen insbesondere Änderungen an der modularen Struktur. Ein Beispiel wäre das Zusammenfassen mehrerer einzelner Module zu einem großen Modul aufgrund logischer Zusammengehörigkeit. Nach der in dieser Arbeit verwendeten Definition handelt es sich – wie bereits erwähnt – bei Anwendungen, die nur solche Transformationen durchführen, nicht um Generatoren[97].

Generatoren, die in einem Schritt sowohl eine Änderung der modularen Strukturen der Eingangsrepräsentation als auch eine Konkretisierung des Modells vornehmen, werden als **Transformationsgeneratoren** bezeichnet.

Die Abb. A.3 verdeutlich die eben eingeführten Begrifflichkeiten. Kompositionsgeneratoren, welche nur auf vertikalen Transformationen basieren, sind einfacher zu implementieren als auf allgemeinen Transformationen basierende. Im Wesentlichen besteht der Transformationsschritt dann auf einem Durchlaufen der gegebenen Repräsentation und dem Ersetzen des jeweiligen Konzeptes durch geeignete Konzepte der niedrigeren Stufe bzw. der Implementation desselben. Außerdem sind die Transformationen im Quellcode leichter nachvollziehbar und die Ergebnisse leichter wartbar, da das Kapselungsprinzip nicht verletzt wird. Generatoren, die einen Transformationsschritt ausführen, welcher die modularen Strukturen des Modells verändert, sind jedoch im Allgemeinen mächtiger. So ist es beispielsweise möglich, Optimierungen auf der Modulebene vorzunehmen. Solche strukturellen Veränderungen[98] am Code erschweren jedoch das Verständnis des Codes auf den unteren Ebenen.

[96] In der Literatur findet sich auch die Bezeichnung Vorwärtsverfeinerung (*forward refinement*)

[97] Dies ist ein Unterschied zu der Betrachtungsweise in [CE00].

[98] Diese strukturellen Änderungen lassen sich weiter unterteilen. Beim Interleaving (engl. Verflechtung, Verschränkung) werden zwei oder mehr Konzepte einer höheren Ebene in einem einzigen Modul der niedrigeren Ebene realisiert. Bei der Delokalisierung wird ein Konzept einer höheren Ebene über mehrere Module der niedrigeren verteilt, d. h. es führt Details an vielen Konzepten der niedrigeren Ebene ein.

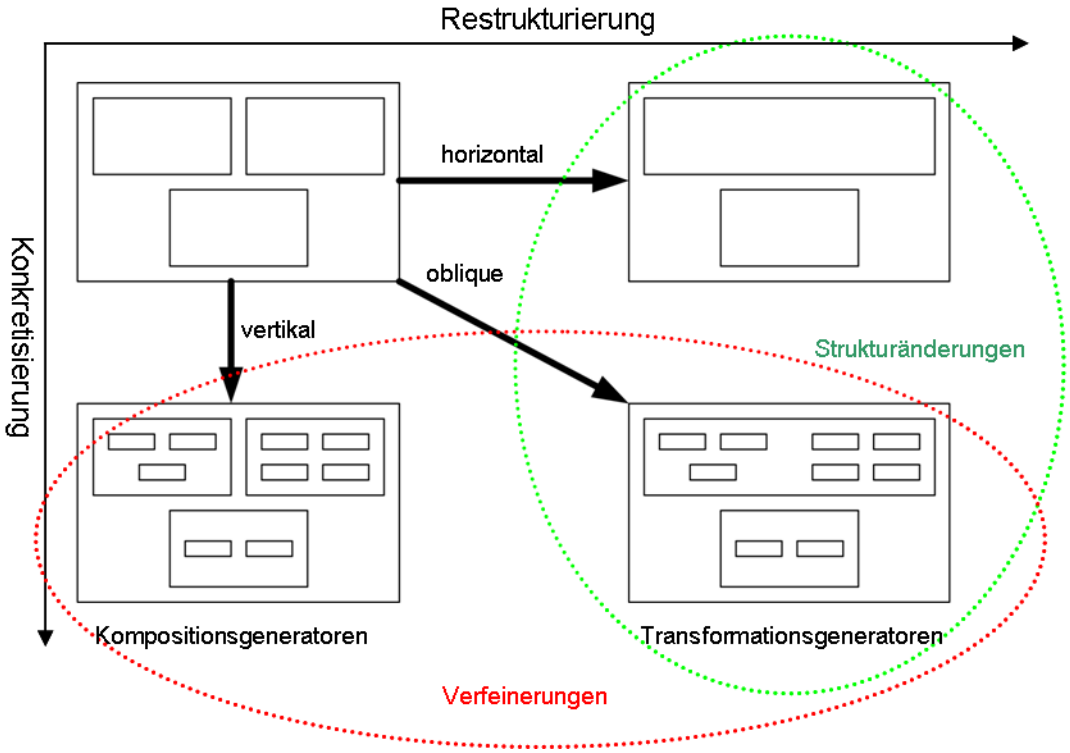


Abb. A.3: Klassifikation von Generatoren (überarbeitet nach [CE00])

Batory und Smaragdakis [BS99] entwickeln eine von der obigen grundverschiedene Klassifikation für Generatoren. Zur Einteilung werden insgesamt vier Eigenschaften von Generatoren betrachtet. Zuerst wird unterschieden, ob es sich um eine eigenständige Anwendung oder um eine Realisierung auf Basis eines allgemeineren Transformationssystems handelt. Die Art der Realisierung der Transformationen wird unterschieden in musterbasierte und programmbasierte Transformationen. Musterbasierte Transformationen werden in einer speziellen Sprache beschrieben und basieren auf dem wiederholten Durchsuchen der Eingabe und Ersetzen gefundener Ausprägungen der jeweiligen Muster. Als programmbasiert werden Manipulationen durch beliebige Programme angesehen. Die Autoren räumen hierbei ein, dass viele Sprachen zur Beschreibung von Transformationen nur schwer als eindeutig muster- oder programmbasiert einzuordnen sind. Das dritte zur Unterscheidung herangezogene Charakteristikum ist die verwendete interne Zwischenrepräsentation des Generators. Diese kann einerseits eine direkte Abstraktion der Syntax sein (syntaxgesteuert) oder andererseits des Kontrollflusses (flussgesteuert) des zu generierenden Programms. Als viertes und letztes Kriterium wird die Anzahl der in jedem Transformationsschritt potenziell anwendbaren Transformationen als Maß für die Komplexität betrachtet. Dabei wird aufgrund der Erfahrungen der Autoren eine Trennung in Systeme mit höchstens zehn möglichen Transformationen pro Schritt (allein stehende Generatoranwendungen mit konkreter Eingabespezifikation) und anspruchsvolleren Systemen, die abstraktere Beschreibungen umsetzen (bis zu mehreren hundert Auswahlmöglichkeiten pro Transformationsschritt), getroffen. Eine solche Unterscheidung über feste Anzahlen kann natürlich nur eine gewisse Orientierung hinsichtlich der Komplexität geben.

Beide Klassifikationsansätze sind unabhängig voneinander. Sie können ergänzend eingesetzt werden, wenn der Detaillierungsgrad der Klassifikation dies nötig macht.

Nachdem nun die Grundlagen für domänenspezifische Sprachen und Generatoren betrachtet wurden, soll im Folgenden der bei der im Rahmen dieser Arbeit vorgenommen prototypischen Implementation

eines Programmiermodells verwendete Ansatz zur Realisierung eines Generators motiviert werden. Dazu werden zunächst einige Anforderungen an Generatoren bzw. ihre Implementation aufgestellt.

Anforderungen an die Implementation von Generatoren

In [Cle88] findet sich die Aussage, dass „Generatoren dazu tendieren Programme zu schreiben, die kein Mensch von Hand schreiben würde“[99]. Im Detail werden die schlechte Formatierung, monolithische Prozeduren, schlechte Variablennamen und noch einige andere Punkte angeführt, die zu einer schlechten Lesbarkeit des generierten Codes führen. Dies führt dazu, dass der generierte Code ohne den erzeugenden Generator nicht oder nur schlecht wartbar ist. Darüber hinaus sind Fehler im Generator dementsprechend schwierig zu finden, da hierzu ein Vergleich der Eingabedaten und der resultierenden Ausgabe unerlässlich ist.

Um diesen und ähnlichen Problemen entgegenzuwirken, kann man eine Reihe von Anforderungen an den Generatorentwurf stellen (siehe auch [Sma04]). Diese werden zunächst diskutiert, bevor im Anschluss einige Implementationstechniken vorgestellt werden.

Falls die Eingabe eines Generators zum Teil aus Code der Zielsprache und zum Teil aus Anweisungen zum Generieren von Code in der Zielsprache besteht, so sollten diese Teile getrennt sein. Die Anweisungen der Eingabesprache bzw. die zu verarbeitende Spezifikation können z. B. in einer separaten Datei gespeichert sein oder im nichtgenerierten Quelltext als Kommentare vorliegen. Das Ziel dieser Anforderung ist es, dass der nichtgenerierte Teil der Software auch von einem Compiler übersetzt werden kann, der die generatorspezifischen Anweisungen nicht kennt.

Der generierte Code sollte durch die Kapselungskonzepte (z. B. Module, Klassen, Prozeduren) der Zielprogrammiersprache klar vom Rest des Quellcodes abgegrenzt werden. Diese Anforderung zielt auf eine Verringerung der Abhängigkeit von einem speziellen Generatorsystem. Idealerweise ist der generierte Code so gekapselt, dass er austauschbar ist, ohne an anderen Stellen der Anwendung Änderungen zu erfordern.

Um die Wartbarkeit des generierten Codes zu erhöhen, sollte dieser gut formatiert sein, sich an eventuelle Formatierungs- und Benennungsrichtlinien der Zielsprache oder auch der anwendenden Organisation halten und die Idiome der Zielsprache nutzen.

Der letzte Punkt zielt auf die Wartbarkeit des Generators selbst ab. Idealerweise ist der Generator mit einem Standardwerkzeug entwickelt, offen und konfigurierbar und von seinen Nutzern anpassbar. Dieser Punkt kann bei einem kommerziellen Produkt mit den Geschäftsinteressen des Generatorherstellers konkurrieren.

Implementation eines Generators

Nachdem in den vorangegangenen Abschnitten ein Überblick über domänenspezifische Sprachen und Generatoren gegeben wurde, wird im Folgenden der im Rahmen dieser Arbeit zur Realisierung des Prototyps verwendete Ansatz zur Implementation eines Generators motiviert. Daher beschränkt sich die Auswahl der präsentierten Techniken auf die für diesen Zweck relevanten.

Die Beispiele werden in Ruby[100] entwickelt; diese Sprache erlaubt eine sehr prägnante Formulierung und bietet eine gute Unterstützung für den Umgang mit regulären Ausdrücken und die Manipulationen

[99] *„Generators tend to create programs that people would not normally write by hand.“*

[100] Ruby ist eine objetkorientierte Skriptsprache, siehe [Rub06] [TFH05].

von Zeichenketten. Der in Ruby generierte Code kann, da es sich um eine interpretierte Sprache handelt, ohne Kompilation ausgeführt werden.

Der einfachste Weg, Code zu generieren, ist die direkte, unparametrisierte Ausgabe durch entsprechende Methoden in eine Datei. Dies entspricht dem weiter oben angesprochenen Generatortyp, welcher nur die leere Eingabe zulässt. Das Codebeispiel A.1 zeigt die Generierung eines Programms, welches den Text *Hallo Welt* auf die Standardausgabe ausgibt. Dieser Ansatz ist jedoch unflexibel und schwerfällig, die Methoden zur Codeausgabe sind darüber hinaus schwer wartbar. Eventuelle Anpassungen erfordern Änderungen am Quelltext des Generatorsystems und gegebenenfalls ein erneutes Übersetzen des Systems.

```
(1)  def simpleCodeGen
(2)    code = "print \"Hallo Welt\""
(3)    File::open("hallo.rb","w") do |f|
(4)      f<<code
(5)    end
(6)  end
(7)
(8)  simpleCodeGen
```

Codebeispiel A.1: Einfacher Generator in Ruby

Um mehr Flexibilität zu erreichen, kann man die zur Codeerzeugung verwendeten Methoden parametrisieren. Im obigen Beispiel könnte man die Methode `simpleCodeGen` mit einem Parameter versehen, der festlegt, welcher Text durch das generierte Programm auf der Standardausgabe ausgegeben werden soll. Beide Varianten der Methode können im Rahmen eines Programms zur Codegenerierung genutzt werden. Um sie interaktiv nutzen zu können wäre ebenfalls ein entsprechendes Rahmenprogramm zur Kommunikation mit dem Nutzer notwendig. Dieses Vorgehen zur Codeerzeugung wird in [Voe03] als API-basierte Codeerzeugung bezeichnet. Kennzeichnend ist hierbei, dass die Methoden im Allgemeinen auf Basis der Abstraktionen des zu erzeugenden Codes beschrieben werden. Es existiert kein Metamodell für eine Eingabespezifikation oder Ähnliches.

Der CodeDOM (Code Document Object Model) Ansatz, den die Firma Microsoft im .NET-Framework realisiert, ist ein Beispiel für die API-basierte Codeerzeugung. Allerdings ist CodeDOM durch die Verwendung eines abstrakten Codemodells sehr viel mächtiger, als es obige Ausführungen vermuten lassen. Mit dem bereitgestellten API lässt sich ein abstrakter Syntaxbaum aufbauen, dieser kann dann in einer Sprache des .NET-Frameworks ausgegeben werden. Direkt durch das Framework unterstützt wird die Ausgabe in C#, J# und VisualBasic, hier sind jedoch durch die modulare Architektur Erweiterungen möglich. Die Granularität der zum Aufbau des Syntaxbaumes benötigten Anweisungen führt zu einer hohen Komplexität beim Einsatz dieser Technik. Da CodeDOM alle Sprachen des .NET-Frameworks unterstützen soll, beschränken sich die unterstützten Sprachfeatures auf die Untermenge der in allen Sprachen verfügbaren Konstrukte. In [Dol04] findet sich eine Aufstellung der unterstützten bzw. nicht unterstützten Sprachkonstrukte für C# und VisualBasic.

Von Microsoft wird ein weiteres API zur Codeerzeugung im .NET-Framework unterstützt. Im `System.Reflection.Emit` [Mic06m] Namensraum finden sich Methoden zur Erzeugung und Ausgabe von Code in der Microsoft Intermediate Language (MSIL). Die Befehle dieser Sprache können – wie in Kapitel 5 beschrieben – direkt von der virtuellen Maschine des .NET-Frameworks, der CLR (Common Language Runtime), ausgeführt werden.

Die API-basierte Codeerzeugung kann in nahe liegender Weise als Basis für komplexere Ansätze, beispielsweise die Umsetzung einer domänenspezifischen Sprache, verwendet werden. Hier ist eine

Grammatik notwendig, die diese Sprache beschreibt. Zusätzlich muss festgelegt werden, wie die Elemente der Eingangssprache auf Konstrukte der Zielsprache abgebildet werden. Die konkrete Realisierung dieser Abbildungen im Generator kann unter Nutzung der API-basierten Codeerzeugung erfolgen. Die Methoden, welche die Abbildung realisieren, rufen ihrerseits Methoden auf, die bestimmte Konstrukte der Zielsprache erzeugen. Als natürliche Erweiterung des „Hallo Welt“-Generators aus Codebeispiel A.1 wird im Folgenden ein einfacher Generator zur Realisierung einer DSL für die Ausgabe formatierter Zeichenketten gezeigt. Die Sprache besteht aus Ausdrücken der Form `{Befehl:Parameter}`. Die Befehle steuern, in welcher Form die als Parameter übergebene Zeichenkette später von dem generierten Programm ausgegeben werden. Eine Eingabe der Form `{upper:hallo}{reverse:welt}` würde ein Programm generieren, welches bei seiner Ausgabe *HALLO tlew* auf der Standardausgabe ausgibt. Der Code ist so strukturiert, dass der Teil, der der oben angesprochenen API entspricht, deutlich vom Rest getrennt ist.

```
(1)   class SimpleDSL
(2)     @@out = 'out.rb'
(3)     @@symbols = {
(4)       'reverse' => :reverse,
(5)       'upper' => :upper,
(6)       'lower' => :lower,
(7)       'plain' => :plain
(8)     }
(9)     def initialize   #Initialisieren von Instanzvariablen
(10)      @code="#generated code\n"
(11)      @rgx = Regexp.new('\{\w+\:[^}]*\}')
(12)      @token = []
(13)    end
(14)    def readInput   #Eingabe lesen und in Token {Befehl:Parameter} zerlegen
(15)      input = gets.strip
(16)      @token = input.scan(@rgx)
(17)      dslParser
(18)    end
(19)    #Token in Befehls- und Parameterteil spalten und die entsprechenden
        #Generierungsmethoden aufrufen
(20)    def dslParser
(21)      @token.each do |token|
(22)        parts = token.split(':')
(23)        parts[0] = parts[0].gsub('{','')
(24)        parts[1] = parts[1].gsub('}','')
(25)        if @@symbols[parts[0]]
(26)          self.send @@symbols[parts[0]], parts[1]
(27)        else       #Syntaxfehler behandeln
(28)          print "\nSyntaxfehler: "+parts[0]+" ist kein Befehl\n"
(29)        end
(30)      end
(31)      writeCode
(32)    end
(33)    def writeCode   #generierten Code in Datei schreiben
(34)      File::open(@@out,"w") do |f|
(35)        f<<@code
(36)      end
(37)    end
(38)    def reverse(args)   #Methoden um Code zu erzeugen, Abbildung auf API
(39)      putPrint(args.reverse)
(40)    end
(41)    def upper(args)
(42)      putPrint(args.upcase)
(43)    end
(44)    def lower(args)
(45)      putPrint(args.downcase)
(46)    end
(47)    def plain(args)
(48)      putPrint(args)
(49)    end
(50)    def putPrint(args)   #eigentliches API, diese Methode generiert Code
(51)      @code.concat("print \" "+args+" \"\n")
(52)    end
(53)  end
(54)  dsl = SimpleDSL.new
(55)  dsl.readInput
```

Codebeispiel A.2: Realisierung einer einfachen DSL in Ruby

Die durch den in Codebeispiel A.2 gezeigten Generator realisierte DSL erlaubt kein Verschachteln von Befehlen in der Form, dass der Rückgabewert eines Befehls als Eingabe eines anderen Befehls verwendet wird. Auch Abhängigkeiten der Befehle untereinander oder Ähnliches sind nicht implementiert, um den Implementationsaufwand des Beispiels nicht unnötig zu erhöhen.

Im obigen Beispielgenerator musste die gesamte Funktionalität zum Parsen der Eingabe, zur Mustererkennung und schließlich zur Ausgabe des generierten Codes implementiert werden. Erweiterungen oder Änderungen der Eingangssprache erfordern gegebenenfalls Anpassungen an unterschiedlichen Stellen des Generatorprogramms.

Daher liegt es nahe, durch Wiederverwendung vorhandener Werkzeuge oder Technologien den Aufwand zu verringern. Eine Möglichkeit besteht hier in der Verwendung der Extensible Markup Language (XML) als Basis der Eingabesprache. Es existiert eine Vielzahl von Werkzeugen in diesem Bereich, ein großer Teil davon ist frei verfügbar. Diese Vorteile werden allerdings mit dem Nachteil der Beschränkung der Syntax der Eingangssprache auf ein XML-konformes Format erkauft.

Die Extensible Stylesheet Language for Transformations (XSLT) ist eine Sprache zur Transformation von XML-Dokumenten in andere XML-Dokumente und liegt seit November 1999 als Empfehlung des World Wide Web Consortiums (W3C) vor. XSLT wurde nicht als generelle XML-Transformationssprache entworfen, sondern soll hauptsächlich die für die Repräsentation mittels der Extensible Stylesheet Language (XSL) notwendigen Transformationen abdecken [Cla99]. Es ist jedoch prinzipiell möglich, ein beliebiges Ausgabedokument zu erzeugen, so lange es sich um ein textbasiertes Format handelt. Dies gilt auch, wenn die Ausgabe kein gültiges XML ist. Zu diesem Zweck wird durch einen XSLT-Prozessor ein so genanntes XSLT-Stylesheet auf ein XML-Eingabedokument angewendet.

Die Verwendung einer Kombination aus XML-basierten domänenspezifischen Sprachen mit einer auf der Transformation mittels XSLT-Stylesheets beruhenden Codegenerierungsarchitektur bietet eine Reihe von Vorteilen. Insbesondere erlaubt dieser Ansatz ein schnelles Entwickeln von neuen domänenspezifischen Sprachen bzw. erleichtert die Anpassbarkeit vorhandener Sprachen. Er erlaubt darüber hinaus die Änderung der DSL, ohne dass eine Anpassung der zur Codegenerierung – in diesem Fall dem Anwenden des Stylesheets und der Ausgabe bzw. Weiterverarbeitung des resultierenden Codes – verwendeten Programmteile notwendig wird. Damit ist eine Konfiguration nach dem Übersetzungszeitpunkt der Generatoranwendung, eventuell sogar durch den Endanwender, möglich. Dies ist eine der Forderungen aus dem Abschnitt Anforderungen an die Implementation von Generatoren auf Seite 203. Zudem entfällt die in [DK98] als Nachteil des Einsatzes domänenspezifischer Sprachen genannte Notwendigkeit, zur Anpassung einer DSL Wissen über Compilertechnologie zu besitzen.

Problematisch ist die Rückmeldung von Fehlern an den Benutzer, diese treten in der Regel erst beim Übersetzen der Zielsprache auf und sind zielsprachenspezifisch. Dies erschwert das Verständnis für Nichtprogrammierer.

Das folgende Codebeispiel A.3 zeigt ein XSLT-Stylesheet zur Realisierung einer domänenspezifischen Sprache auf XML-Basis. Die Sprache verfügt über dieselben Möglichkeiten, wie die in Ruby implementierte (siehe Codebeispiel A.2), allerdings ist die Syntax nun XML-konform. Die oben angesprochene Beispieleingabe hätte nun die Form `<input><upper>hallo</upper> <reverse>welt</reverse></input>`. Durch Anwenden des Stylesheets auf diese Eingabe wird ein Ruby-Programm generiert, welches ebenfalls *HALLO tlew* auf der Standardausgabe ausgibt.

```
(1)  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
(2)     <!--Keine XML-Deklaration in Output schreiben -->
(3)     <xsl:output method="text"/>
(4)     <xsl:template match="/">
(5)        #generated code for Ruby
(6)        <xsl:apply-templates />
(7)     </xsl:template>
(8)     <xsl:template match="upper">
(9)        <xsl:apply-templates />
(10)       print " <xsl:value-of select="."/> ".upcase
(11)    </xsl:template>
(12)    <xsl:template match="lower ">
(13)       <xsl:apply-templates />
(14)       print " <xsl:value-of select="."/> ". downcase
(15)    </xsl:template>
(16)    <xsl:template match="reverse">
(17)       <xsl:apply-templates />
(18)       print " <xsl:value-of select="."/> ".reverse
(19)    </xsl:template>
(20)    <xsl:template match="plain">
(21)       <xsl:apply-templates />
(22)       print " <xsl:value-of select="."/> "
(23)    </xsl:template>
(24)    <!--Standardregel fuer Textknoten ueberschreiben-->
(25)    <xsl:template match="text()"></xsl:template>
(26) </xsl:stylesheet>
```

Codebeispiel A.3: XSLT-Stylesheet zur Realisierung einer DSL

Durch ein entsprechendes XSLT-Stylesheet ließe sich ohne Anpassungen am verwendeten Codegenerator auch Dokumentation zu einem DSL-Programm erzeugen.

A.6 Vor- und Nachteile von Codegeneratoren

Das Generieren von Code bringt eine Reihe von Vorteile mit sich (vgl. auch [Her03]), es existieren aber auch einige Nachteile. Im Folgenden wird ein kurzer Überblick hierüber gegeben.

Der Einsatz eines Generators kann die Konsistenz und Lesbarkeit der Codebasis eines Projektes verbessern. Wenn die Regeln zur Codeerzeugung korrekt implementiert wurden, dann erzeugt der Generator immer Code, der sich an die Benennungs- und Formatierungsrichtlinien des Projektes hält. Außerdem werden für sich wiederholende Aufgaben immer wieder die gleichen Lösungsansätze verwendet. Wird beispielsweise ein Generator zum Erzeugen der Zugriffsfunktionalität auf die Daten einer relationalen Datenbank verwendet, so wird der zugehörige Code an allen Stellen im Projekt – unabhängig vom Entwickler, der für diesen speziellen Codeabschnitt verantwortlich ist – immer gleich aussehen und funktionieren.

Der letzte Punkt führt zu weiteren Vorteilen. Falls eine Änderung am generierten Code nötig ist, sei es aufgrund einer Änderung der Anforderungen oder um einen Fehler zu beheben, so wird dies im Falle eines Generators durch eine Änderung an den Erzeugungsregeln des Generators erreicht. Unabhängig davon, an wie vielen Stellen des Projektes der betroffene Code verwendet wird. Ohne Einsatz eines Generators müssen diese Änderungen an allen Stellen von Hand ausgeführt werden. Im oben erwähnten Beispiel eines generierten Datenbankzugriffs würden sich bei einem Wechsel der zugrunde liegenden Datenbank die notwendigen Änderungen auf eine Stelle, den Generator, beschränken.

Im Zusammenhang mit domänenspezifischen Sprachen wurde bereits die Möglichkeit einer Trennung von Spezifikation der Anwendungslogik und konkreter Implementation angesprochen. Aus einer implementationsunabhängigen Beschreibung kann Code für unterschiedliche Plattformen erzeugt werden. Daneben können aus einer Beschreibung unterschiedliche Artefakte generiert werden, denkbar ist z. B. die Dokumentation des erzeugten Codes mitzugenerieren.

In [CE00] wird angesprochen, dass Generatoren dazu beitragen, das von Biggerstaff in [Big94] dargelegte „Library Scaling Problem“ zu verhindern (siehe auch [Big98]). Biggerstaff führt aus, dass es für den Ersteller einer Bibliothek erstrebenswert ist, möglichst große Komponenten zu implementieren, da die Kombination einiger weniger großer Komponenten einfacher ist als die vieler kleiner (vertikale Skalierung). Je umfangreicher jedoch die Funktionalität einer Komponente wird, desto spezifischer wird sie und desto schwieriger ist ihre Wiederverwendung. Um eine Komponentenbibliothek mit möglichst häufig und in vielen Anwendungen wieder verwendbaren Komponenten zu erstellen, muss eine Vielzahl von Datenstrukturen und Funktionalitäten bereitgestellt werden (horizontale Skalierung). Das aus den angesprochenen Motivationen resultierende Bestreben, eine Bibliothek sowohl horizontal als auch vertikal zu skalieren, erfordert die Implementation einer Vielzahl von Kombinationen der durch die Bibliothek bereitgestellten Eigenschaften in Form von konkreten Komponenten. Dies führt beim Hinzufügen neuer Features zu einem exponentiellen Wachstum der Bibliothek[101]. Die Alternative, die bereitgestellten Eigenschaften in Komponenten zu kapseln und durch Methodenaufrufe wieder zu verwenden, führt laut Czarnecki zu Performanzproblemen. Als Lösung wird die Erzeugung von Komponenten, welche die benötigten Eigenschaftskombinationen bereitstellen, mittels Generatoren angeführt.

Nachteile entstehen durch den nötigen Entwicklungsaufwand für einen Generator. In [BDG+95] wird ein um 50 % bis 100 % höherer Aufwand bei der Entwicklung eines Generators für ein Softwaresystem im Vergleich mit der direkten Entwicklung eines Einzelsystems angenommen.

Darüber hinaus wird durch die Verwendung eines Generators ein zusätzliches Werkzeug in den Entwicklungsprozess eingeführt, mit dem die Entwickler vertraut sein müssen. Wenn der Generator keine Eigenentwicklung sondern ein gekauftes Werkzeug ist, so resultiert sein Einsatz unter Umständen in einem Verlust an Kontrolle über gewählte Datenstrukturen und ihre interne Realisierung, über die Implementation gewählter Algorithmen und über Formatierungs- und Benennungsrichtlinien für den generierten Code. Fehler, die während des Generierungsprozesses entstehen, sind nur schwer nachvollziehbar.

[101] Als Beispiel für diesen Effekt wird eine von Grady Booch in ADA erstellte Bibliothek für Datenstrukturen angeführt.

Appendix B: Definitionen des Begriffs Softwarekomponente

Der Beginn der Weisheit ist die Definition der Begriffe.
Sokrates, griechischer Philosoph, 470 - 399 v. Chr.

In diesem Anhang sind einige der bei den Recherchen und Überlegungen zu einem konzeptionellen Komponentenbegriff gefundenen Definitionen für den Begriff Softwarekomponente aufgeführt. Die Zitate sind in chronologisch aufsteigender Ordnung nach den Jahreszahlen (von 1996 bis 2006) angegeben. Die Definitionen werden in ihrer ursprünglichen Form wiedergegeben, Auslassungen sind durch „[...]“ gekennzeichnet.

1. Definition der WCOP'96

Die im Rahmen der WCOP'96 erarbeiteten Definition wird häufig angeführt, so auch bei [BSW02] und [SGM03]:

"A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Components can be deployed independently and are subject to composition by third parties."

2. Definition von Ousterhout, 1997

In [Ous97] findet sich keine vollständige Definition für den Begriff Komponente, er wird aber (insbesondere im Zusammenhang mit Skriptsprachen) wie folgt gebraucht:

"[…] scripting languages are designed for gluing: they assume the existence of a set of powerful components and are intended primarily for connecting components together."

"Components are designed to be reusable, and there are well-defined interfaces between components and scripts that make it easy to use components."

3. Definition von Nierstrasz und Lumpe, 1997

In [NL97] wird Folgendes zum Komponentenbegriff ausgeführt:

„Komponenten definieren weder einen neuen Objektbegriff, noch ersetzen sie Objekte in ihrer bisherigen Bedeutung. Unter Komponenten sollten wir vielmehr eine neue Art und Weise verstehen, mit der man objektorientierte Softwareentwicklung betreibt. Mit anderen Worten, eine gut strukturierte, flexible, objektorientierte Applikation ist auch komponentenorientiert. [...] Eine Komponente kann daher auch nicht losgelöst vom ihrer Umgebung betrachtet werden. [...] Eine "Softwarekomponente" ist ein "Element eines Komponentenframeworks". [...] Eine Softwarekomponente ist eine "statische Softwareabstraktion mit Plugs". Unter einer Softwareabstraktion verstehen wir mehr oder weniger eine sogenannte "Black- Box". Die Haupteigenschaft dieser Black-Box ist, daß sie die Implementationspezifikation vor dem Benutzer verbirgt. Die Eigenschaft "statisch" bedeutet in diesem Zusammenhang, daß wir diese "Black-Box" in einer geeigneten Weise instanziieren müssen, bevor wir sie benutzen können. In objektorientierten Programmiersprachen (sofern sie die Abstraktion Klasse unterstützen) unterscheiden wir zwischen "Klassen" und "Objekten", wobei Objekte Instanzen einer Klasse sind. Leider gibt es für Komponenten keine derartige Unterscheidung. Wenn wir mit Komponenten arbeiten, benutzen wir meistens Instanzen von Komponenten. [...] Unter Plugs verstehen wir die Softwareschnittstelle einer Komponente [...] Es gibt keine versteckten Abhängigkeiten. [...] Der entscheidende Punkt ist, daß wir Komponenten nicht

isoliert betrachten können. Man muß sie immer im Zusammenhang mit der ihnen zugrunde liegenden Softwarearchitektur des Komponentenframeworks sehen.“

4. Definitionen aus Software – Concepts&Tools 1998

Die folgenden Definitionen entstammen einer per E-Mail geführten Diskussionsrunde, die in [BDH+98] niedergelegt ist. Im Folgenden werden einzelne Vorschläge der Beteiligten angegeben.

Kai Koskimies:

“A component is a system-independent binary entity which implements one or more interfaces. An interface is a collection of signatures of services belonging logically together.”

Wolfgang Pree, Gustav Pomberger:

“Our definition of components is derived from examining the deficiencies of the object-oriented paradigm:

- Classes/objects implemented in one programming language cannot interoperate with those implemented in other languages.
- Objects are typically composed on the language level. Black-box composition support is missing, that is, visual/interactive tools that allow the plugging together of objects.

Characteristics of components:

- A component is simply a data capsule. Thus information hiding becomes the core construction principle underlying components.
- A component can be implemented in (almost) any language, not only in any module-oriented or objectoriented languages but even in conventional languages.
- As a consequence, standards for describing components have been established.
- The component (module) interface is described either
  - textually by means of an interface description language (IDL) or
  - visually/interactively using appropriate tools.
- Framework architectures form the enabling technology of plug & play software, where most adaptations can be achieved by exchanging components. Visual, interactive composition tools are ideally built on top of domain-specific frameworks.
- Single-component reuse is less attractive. It means that programmers build the overall software system architecture on their own. They have to locate the appropriate components in a Lego building block library and define their interactions.”

Michael Stal:

Es wurden drei Definitionen vorgeschlagen, wobei die erste deutlich näher an technischen Gegebenheiten orientiert ist:

“A component is a binary unit that exports and imports functionality using a standardized interface mechanism. The underlying component infrastructure supports composition of components by providing mechanisms for introspection, eventhandling, persistence, dynamic linking and layout

management. In general, application frameworks are required for building components as well as for composing them."

"A component is a binary software module that exports and imports functionality using a standardized interface mechanism."

"A component denotes a self-contained entity (black box) that exports functionality to its environment and may also import functionality from its environment using well-defined and open interfaces. In this context, an interface defines the syntax and semantics of the functionality it comprises (i.e., it defines a contract between the environment and the component). Components may support their integration into the surrounding environment by providing mechanisms such as introspection or configuration functionality."

5. Definition von Bergner, Rausch, Sihling und Vilbing, 1998

In [BRS+98] wird die folgende Definition für den Begriff Komponente gegeben:

"A component provides *export interfaces* to offer specific services to other components and it uses *import interfaces* to access services provided by other components. In addition, there may be *combined interfaces* combining the properties of export and import interfaces."

6. Definition von D'Souza, 1999

D'Souza beschreibt Komponenten in [DW99] auf Seite 386 wie folgt:

"A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged , with other components to build something larger."

7. Definition von Sparling, 2000

Sparling verwendet in [Spa00] die Definition:

"A component is a language neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interfaces. While a component may have the ability to modify a database, it should not be expected to maintain state information. A component is not platform-constrained nor is it application-bound."

8. Definition von Gruhn und Thiel, 2000

In [GT00] verwenden Gruhn und Thiel folgende Definition:

„Eine Komponente ist ein Stück Software in binärer Form, das eine kohärente Funktionalität bietet. Die strikte Kapselung der Implementierung und die damit verbundene Black Box-Wiederverwendung führt zu einer gewissen Eigenständigkeit der Komponenten und ermöglicht somit eine lose Kopplung zwischen der Komponente und ihrer Umgehung. Die angebotene Funktionalität wird mittels einer oder mehrerer Schnittstellen beschrieben.

Es wird zwischen dem Begriff Komponente, der die statische Beschreibung einer Komponente (vergleichbar mit der Klasse im objektorientierten Paradigma) bezeichnet, und dem Begriff Komponenteninstanz (vergleichbar mit dem Objekt im objektorientierten Paradigma) unterschieden. Eine Komponenteninstanz kann über eine persistente (vgl. Persistenz) Identität verfügen, so dass sie auch über die Lebensdauer des sie erzeugenden Prozesses hinaus eindeutig referenziert werden kann.

Neben dem Binärcode kann eine Komponente weitere zugehörige Artefakte enthalten, die während der Anwendungsentwicklung oder zur Laufzeit hilfreich bzw. notwendig sind. Dazu zählen beispielsweise Dokumentation, selbstbeschreibende Metadaten für die Introspektion, Stub- und Skeleton-Klassen zur Verteilung, Factories zur Erzeugung von Instanzen, Hilfsklassen, sowie Serialisierungsinformationen.

Eine Komponente ist immer für die Verwendung innerhalb genau eines konkreten Komponentenmodells ausgelegt. Der Begriff Komponentenmodell wird wie folgt interpretiert:

Ein Komponentenmodell legt einen Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten (Komposition) sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten an die Komponenten stellt. Darüber hinaus wird durch ein Komponentenmodell eine Infrastruktur angeboten, die häufig benötigte Mechanismen wie Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung implementieren kann.“

9. Definition von Reinhardt, 2000

[Rei00] beschreibt eine Komponente wie folgt:

„Eine Softwarekomponente...

1. repräsentiert einen *abgeschlossenen* Dienst hinsichtlich eines spezifischen, *anwendungs-orientierten* Nutzungsinteresses (vgl. Abschnitt 3.1.1).
2. kann *einfach* verwendet werden, das heißt ohne dass (aus Sicht des Nutzungsinteresses) aufwendige Konfigurationen nötig sind (vgl. Abschnitt 3.1.1).
3. besitzt explizite Schnittstellen zu dem von ihr angebotenen Dienst (vgl. Abschnitt 3.1.1).
4. kann weitestgehend unabhängig von der Programmiersprache verwendet werden, in der sie entwickelt wurde (vgl. Abschnitt 3.1.2).
5. kann dynamisch, das heißt zur *Laufzeit* an in Ausführung befindlichen Code gebunden werden.“

10. Definition von Ambler, 2001

Ambler definiert eine Komponente in [Amb01] als:

“A component is a modular, extensible unit of independent deployment that has contractually specified interface(s) and explicitly defined dependencies, if any. Ideally, components should be modular, extensible, and open. Modularity implies a component contains everything it needs to fulfil its responsibilities, extensibility implies that a component can be enhanced to fulfil more responsibilities than it was originally intended to, and open implies that it can operate on several platforms and interact with other components through a single programming interface. “

11. Definition von Clemente, 2002

[CSP02] verwendet (Quellenangabe: [SGM03] in der ersten Auflage von 1998):

„The process of building applications becomes into a process of assembling independent and reusable software modules called components. A component is a software composition unit that specifies a set of interfaces and a set of requirements. A component is thought to be developed, acquired and incorporated into the system. It can also be composed with other components independently in time and space.“

12. Definition vom Second International Workshop on Composition Languages (WCL 2002), Lumpe, Schneider, Schönhage, Genssler, 2002

In [LSS+02] findet sich folgenden Definition:

“Furthermore, we defined the term (software) component for the remainder of the workshop as an abstraction that

- is a self-contained unit of composition,
- is built to be composed (i.e. is an element of a component framework),
- offers a collection of services and requires a collection of (other) services in order to be functional, and
- is an (abstract) entity that needs adaptation in order to be (re-)used.”

13. Definition von Szyperski, 2003

In [SGM03] finden sich neben der Definition der WCOP’96 noch weitere Ausführungen zum Komponentenbegriff:

Im Vorwort zur ersten Auflage heißt es:

“[...] software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.”

Und weiter in Kapitel 4, S. 36:

„The characteristic properties of a component are that it:

- Is a unit of independent deployment;
- Is a unit of third-party composition;
- Has no (externally) observable state.“

und weiter unten:

„[…] a component will never be deployed partially.”

Außerdem führt Szyperski aus, dass es aufgrund des Fehlens eines von Außen beobachtbaren Zustandes keinen Sinn habe, mehrere Komponenteninstanzen innerhalb eines Kontextes zu haben, da diese nicht unterscheidbar wären.

Im Abschnitt „What others say“ finden sich die nachfolgenden Zitate:

Grady Booch, Software Components with Ada: Structures, Tools, and Subsystems, 1987:

“A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.”

Ivar Jacobsen, Object-Oriented Software Engineering, 1993:

“By components we mean already implemented units that we use to enhance the programming language constructs. These are used during programming and correspond to the components in the building industry. “

Jed Harris, President CI Labs:

“A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability.”

14. Definition der OMG, 2003

Die UML Spezifikation [Obj03a] definiert auf S. 3-174 f. eine Komponente wie folgt:

“A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component is typically specified by one or more classifiers that reside on the component. A subset of these classifiers explicitly define the component’s external interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component. A component may be implemented by one or more artifacts, such as binary, executable, or script files. A component may be deployed on a node.”

Auf S. 2-98 wird der Begriff Komponenteninstanz definiert:

“A component instance is an instance of a component that resides on a node instance. A component instance may have a state.”

15. Definition von Amza und Reggio, 2003

[AR03] verwenden den folgenden Komponentenbegriff:

„A component has to come with a clear specification of the services it provides. Because a component cannot make any assumption about the environment in which it will be used, its definition should specify the services required, such that the component can work.“

„A component is an entity that denotes a collection of cooperating objects and/or components relevant to a particular concept, goal or purpose. A component has three distinct parts: an interface, a body and a context. The component interface defines the services that the component provides to its clients and those that it requires from other components in the system. We extend the notion of service defined in the previous section to denote a named set of related methods and communication channels. The component context defines all the classes used by the services defined in the component interface and makes them visible for all external entities using the component. This is a novel approach that allows other components to use introspection mechanisms in order to discover the services that a component requires/provides. The component body describes how the services provided by the component are implemented, using the required ones, if is the case.“

16. Definition von Sora, Verbaeten und Berbers, 2003

In [SVB03] findet sich:

“Software component: is an implementation of some functionality, available under the condition of a certain contract, independently deployable and subject to composition. A component in our approach is also an architectural abstraction.”

17. Definition von Gill, 2006

Bei [Gil06] wird der Begriff der Komponente wie folgt verwendet:

“A component can be considered an independent replaceable part of the application that provides a clear distinct function. A component can be a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system.”

Literaturverzeichnis

[AR03] **Amza, C. und Reggio, G.:** "A Notation for Component-Based Design of Java Applications", FIGJI 2002, LNCS 2604, Berlin Heidelberg: Springer, pp. 155- 164, 2003.

[Ala85] **Alavi, M.:** "Some thoughts on quality issues of end-user developed systems", SIGCPR '85: Proceedings of the twenty-first annual conference on Computer personnel research, New York, NY, USA: ACM Press, pp. 200-207, 1985.

[All02] **Allowatt, T.:** "Bending the.NET PropertyGrid to Your Will", *http://www.codeproject.com/cs/miscctrl/bending_property.asp,* 2002, besucht am: 17.05.2005.

[Amb01] **Ambler, S. W.:** "The object primer: the application developer's guide to object orientation", Cambridge [England], New York: Cambridge University Press, 2001.

[Auf05] **Aufsichtsbehörde für den Datenschutz im nicht-öffentlichen Bereich des Landes Sachsen-Anhalt:** "2. Tätigkeitsbericht 01.06.2003 - 31.05.2005", Halle, 2005.

[BAD+01] **Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., Yang, S.:** "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm" in Journal of Functional Programming, March 11 (2), pp. 155-206, 2001.

[BB93] **Brancheau, J. C. und Brown, C. V.:** "The management of end-user computing: status and directions" in ACM Computing Surveys (CSUR), 25 (4), pp. 437-482, 1993.

[BC89] **Beck, K. und Cunningham, W.:** "A laboratory for teaching object oriented thinking", OOPSLA '89: Conference proceedings on Object-oriented programming systems languages and applications, New York, NY, USA: ACM Press, pp. 1-6, 1989.

[BCR+00] **Batory, D. S., Chen, G., Robertson, E. und Wang, T.:** "Design Wizards and Visual Programming Environments for GenVoca Generators" in Software Engineering, 26 (5), pp. 441-452, 2000.

[BCR04] **Burnett, M., Cook, C. und Rothermel, G.:** "End-user software engineering" in Communications of the ACM, 47 (9), pp. 53-58, 2004.

[BDG+95] **Batory, D., Dasari, S., Geraci, B., Singhal, V., Sirkin, M. und Thomas, J.:** "Achieving Reuse With Software System Generators", Department of Computer Sciences, University of Texas, Austin, USA, 1995.

[BDH+98] **Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M. und Szyperski, C.:** "What characterizes a (software) component?" in Software - Concepts & Tools, 4 (19), pp. 49-56, 1998.

[BH98] **Brynjolfsson, E. und Hitt, L. M.:** "Beyond the productivity paradox" in Communications of the ACM, 41 (8), pp. 49-55, 1998.

[BJM+02] **Batory, D., Johnson, C., MacDonald, B. und von Heeder, D.:** "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study" in ACM Transactions on Software Engineering and Methodology, Vol. 11 (No. 2), pp. 191-214, 2002.

[BRS+98] **Bergner, K., Rausch, A., Sihling, M. und Vilbig, A.:** "An Integrated View On Componentware – Concepts, Description Techniques, and Development Process", Institut für Informatik, Technische Universität München, München, 1998.

[BS02] **Broy, M. und Siedersleben, J.:** "Objektorientierte Programmierung und Softwareentwicklung - Eine kritische Einschätzung" in Informatik Spektrum, 25 Februar, pp. 3-11, 2002.

[BS99] **Batory, D. und Smaragdakis, Y.:** "Application Generators", Department of Computer Science, University of Texas, Austin, Texas, USA, 1999.

[BSW02] **Bosch, J., Szyperski, C. und Weck, W.:** "Component-Oriented Programming", ECOOP 2002, LNCS 2548, Berlin [et al.]: Springer, pp. 70-78, 2002.

[Bal05] **Balzert, H.:** "Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2", München: Elsevier, Spektrum, Akademischer Verlag, 2005.

[Bal85] **Balzer, R.:** "A 15 Year Perspective on Automatic Programming" in IEEE Transactions on Software Engineering, November SE-11 (11), pp. 1257-1268, 1985.

[Bay04] **Bayerische Datenschutzaufsicht für den nicht-öffentlichen Bereich:** "1. Tätigkeitsbericht 2002/2003", Ansbach, 2004.

[Ben86] **Bentley, J.:** "Little Languages" in Communications of the ACM, 29 (8), pp. 711-721, 1986.

[Ber93] **Bernd, M.:** "Is object-oriented programming structured programming?" in ACM SIGPLAN Notices, 28 (9), pp. 57-66, 1993.

[Big94] **Biggerstaff, T. J.:** "The Library Scaling Problem and the Limits of Concrete Component Reuse", Third International Conference on Software Reuse, Los Alamitos: IEEE Computer Society Press, pp. 102-109, 1994.

[Big98] **Biggerstaff, T.:** "A Perspective of Generative Reuse" in Annals of Software Engineering, (5), pp. 169-226, 1998.

[Bra04] **Braude, E. J.:** "Software design: from programming to architecture", Hoboken, NJ: John Wiley and Sons, 2004.

[Bro96] **Broy, M.:** "Towards a Mathematical Concept of a Component and Its Use - Keynote Speech at the Components' Users Conference CUC'96 Revised and Extended Version", Fakultät für Informatik - Technische Universität München, München, München, 1996.

[Bro98] **Broy, M.:** "Appendix to M. Broy et al.: What characterizes a (software) component?" in Software - Concepts & Tools, 1998 4 (19), pp. 57-59, 1998.

[Bur99] **Burnett, M.:** "Visual Programming" in John G. Webster, eds., Enclopedia of Electrical and Electronics Engineering. Hoboken, NJ: John Wiley and Sons, 1999, pp. 275-283.

[Bus98] **Buschmann, F.:** "Pattern-orientierte Software-Architektur: ein Pattern-System", Bonn [u.a.]: Addison-Wesley-Longman, 1998.

[CA06] **Cwalina, K. und Abrams, B.:** "Framework Design Guidelines", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 2006.

[CB05] **Connolly, T. M. und Begg, C. E.:** "Database systems: a practical approach to design, implementation, and management", Harlow [u.a.]: Addison-Wesley, 2005.

[CDK02] **Coulouris, G. F., Dollimore, J. und Kindberg, T.:** "Verteilte Systeme: Konzepte und Design", München: Pearson Studium, 2002.

[CE00] **Czarnecki, K. und Eisenecker, U.:** "Generative programming: methods, tools, and applications", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 2000.

[CE99] **Czarnecki, K. und Eisenecker, U. W.:** "Components and Generative Programming" in O. Nierstrasz und M. Lemoine, eds., Software Engineering - ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999. Proceedings. Berlin [u.a.]: Springer, 1999, pp. 2-19.

[CK89] **Cotterman, W. W. und Kumar, K.:** "User cube: a taxonomy of end users" in Communications of the ACM, 32 (11), pp. 1313-1320, 1989.

[CM98] **Consel, C. und Marlet, R.:** "Architecturing Software Using A Methodology for Language Development" in C. Palamidessi, H. Glaser und K. Meinke, eds., Principles of Declarative Programming: 10th International Symposium, PLILP'98. Berlin [u.a.]: Springer, 1998, pp. 170-194.

[CSP02] **Clemente, P. J., Sánchez, P. J. und Pérez, M. A.:** "Modeling with UML Component-based and Aspect Oriented Programming Systems", Quercus Software Engineering Group. Extremadura University, Spain, 2002.

[Cha04] **Chatterjee, S.:** "Messaging Patterns in Service-Oriented Architecture, Part 1" in JOURNAL - Microsoft Architect's Journal, 2, April 2004.

[Cla99] **Clark, J. (ed.):** "XSL Transformations (XSLT) Version 1.0 W3C Recommendation", *http://www.w3.org/TR/1999/REC-xslt-19991116*, 1999, besucht am: 02.10.2005.

[Cle03] **Clement, T.:** "A Pretty Good Splash Screen in C#", *http://www.codeproject.com/csharp/PrettyGoodSplashScreen.asp,* 2003, besucht am: 16.05.2006.

[Cle88] **Cleaveland, J. C.:** "Building Application Generators" in IEEE Software, 5 (4), pp. 25-33, 1988.

[Cli03] **Clifton, M.:** "A look at what's wrong with objects", *http://www.codeproject.com/gen/design/theWrongObject.asp,* 2003, besucht am: 18.06.2004.

[Cli06] **Clifton, M.:** "MyXaml - Home", *http://www.myxaml.com/,* 2006, besucht am: 09.07.2006.

[Con04] **Consel, C.:** "From A Program Family To A Domain Specific Language" in C. Lengauer, D. Batory, C. Consel und M. Odersky, eds., Domain-Specific Program Generation. Berlin [u.a.]: Springer, 2004, pp. 19-29.

[Cyp94] **Cypher, A.:** "Watch what I do: programming by demonstration", Cambridge, Mass. [u.a.]: The MIT Press, 1994.

[DK98] **Deursen, A. v. und Klint, P.:** "Little Languages: Little Maintenance?" in Journal of Software Maintenance, 10, pp. 75-92, 1998.

[DKV00] **Deursen, A. v., Klint, P. und Visser, J.:** "Domain-Specific Languages: An annotaded Bibliography" in ACM SIGPLAN Notices, 35 (6), pp. 26-36, 2000.

[DW99] **D'Souza, D. F. und Wills, A. C.:** "Objects, components and frameworks with UML: the catalysis approach", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 1999.

[Dol04] **Dollard, K.:** "Code Generation in Microsoft.Net", Berkeley: Apress, 2004.

[Dot06] **DotGNU Project - GNU Freedom for the Net:** "Free Software for Webservices and for C# Programming", *http://www.dotgnu.org/,* 2006, besucht am: 30.01.2006.

[Dow04] **Downey, J. P.:** "Toward a Comprehensive Framework: EUC Research Issues and Trends (1990-2000)" in Journal of Organizational and End User Computing, Oct-Dez 16 (4), pp. 1-16, 2004.

[ECM05a] **ECMA International:** "Standard ECMA-334, C# Language Specification, 3rd edition", Geneva, Switzerland, 2005.

[ECM05b] **ECMA International:** "Standard ECMA-335, Common Language Infrastructure (CLI), 3rd edition", Geneva, Switzerland, 2005.

[EN05] **Elmasri, R. und Navathe, S.:** "Grundlagen von Datenbanksystemen", München [u.a.]: Pearson Studium, 2005.

[EUS05] **EUSES - End Users Shaping Effective Software:** "End User Programming Errors Resource", *http://eusesconsortium.org/euperrors.php,* 2005, besucht am: 16.01.2006.

[Eif06] **Eiffel Software:** "Eiffel Software - The Home of EiffelStudio and EiffelEnvision", *http://www.eiffel.com/,* 2006, besucht am: 10.03.2006.

[Eur06] **European Spreadsheet Risks Interest Group (EuSpRIG):** "Spreadsheet mistakes - news stories", *http://www.eusprig.org/stories.htm,* 2006, besucht am: 11.01.2006.

[FS97] **Fayad, M. und Schmidt, D. C.:** "Object-oriented application frameworks" in Communications of the ACM, 40 (10), pp. 32-38, 1997.

[Fin05] **Findy Services and Jacobs, B.:** "Object Oriented Programming Oversold", *http://www.geocities.com/tablizer/oopbad.htm,* 2005, besucht am: 13.02.2006.

[Fow04] **Fowler, M.:** "Inversion of Control Containers and the Dependency Injection pattern", *http://www.martinfowler.com/articles/injection.html,* 2004, besucht am: 29.10.2005.

[Fow06] **Fowler, A. (Sun Developer Network):** "A Swing Architecture Overview - The Inside Story on JFC Component Design", *http://java.sun.com/products/jfc/tsc/articles/architecture/index.html,* 2006, besucht am: 23.03.2006.

[GA03] **Griffiths, I. und Adams, M.:** ".NET Windows Forms", Sebastopol, CA, USA: O'Reilly, 2003.

[GP98] **Gruntz, D. und Pfister, C.:** "Komponentensoftware und ihre speziellen Anforderungen an Komponentenstandards" in OBJEKTspektrum, Juli/August (4), pp. 38-42, 1998.

[GT00] **Gruhn, V. und Thiel, A.:** "Komponentenmodelle - DCOM, JavaBeans, Enterprise JavaBeans, Corba", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 2000.

[Gam95] **Gamma, E.:** "Design patterns: elements of reusable object-oriented software", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 1995.

[Gil06] **Gill, N. S.:** "Importance of software component characterization for better software reusability" in SIGSOFT Software Engineering Notes, 31 (1), pp. 1-3, 2006.

[Gov03] **Govindarajulu, C.:** "End users: who are they?" in Communications of the ACM, 46 (9), pp. 152-159, 2003.

[Gri01] **Griffel, F.:** "Verteilte Anwendungssysteme als Komposition klassifizierter Softwarebausteine: Ein Komponentenbasierter Ansatz zur Generativen Softwarekonstruktion", Fachbereich Informatik der Universität Hamburg, 2001.

[Gru00] **Grune, D.:** "Modern compiler design", Chichester [u.a.]: Wiley, 2000.

[HH97] **Hopkins, T. und Horan, B.:** "Objektorientierte Programmierung mit Smalltalk: Einführung in die Applikationsentwicklung mit VisualWorks", München [u.a.]: Hanser [u.a.], 1997.

[HK04] **Hauser, R. und Koehler, J.:** "Compiling Process Graphs into Executable Code", GPCE 2004, LNCS 3286, Berlin Heidelberg: Springer, pp. 317 - 336, 2004.

[HKS04] **Holm, C., Krüger, M. und Spuida, B.:** "Dissecting a C# Application - Inside SharpDevelop", Berkeley: Apress, 2004.

[HP02] **Hobbs, V. J., Pigott, D. J.:** "Facilitating End User Database Development by Working with Users' Natural Representation of Data" in Tonya Barrier, eds., Human computer interaction development and management. Hershey: IRM Press, 2002.

[Har00] **Harris, R.:** "Schools of thought in research into end-user computing success" in Journal of Organizational and End User Computing, Jan-Mar 12 (1), pp. 24-34, 2000.

[Har04] **Harrison, W.:** "The Dangers of End-User Programming" in IEEE Software, Jul-Aug 21 (4), pp. 5-7, 2004.

[Her03] **Herrington, J.:** "Code generation in action", Greenwich, CT: Manning, 2003.

[IBM03] **IBM:** "Websphere MQ - Application Programming Guide", 2003.

[IBM06] **IBM:** "WebSphere MQ", *http://www-306.ibm.com/software/integration/wmq/,* 2006, besucht am: 23.06.2006.

[JFF+02] **Jorgensen, P., Fernandez, D., Fischer, A. und et.al.,:** "Has the Object-Oriented Paradigm Kept Its Promise?", Department of Computer Science and Information Systems, Grand Valley State University, Allendale, MI, USA, 2002.

[JH02] **Jähnichen, S. und Herrmann, S.:** "Was, bitte, bedeutet Objektorientierung?" in Informatik Spektrum, 8. August, pp. 266-276, 2002.

[Jos03] **Josefsson, S. (ed.):** "Request for Comments: 3548: The Base16, Base32, and Base64 Data Encodings", Network Working Group, IETF, 2003.

[KHB+05] **Kandogan, E., Haber, E., Barrett, R., Cypher, A., Maglio, P. und Zhao, H.:** "A1: end-user programming for web-based system administration", UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology, New York, NY, USA: ACM Press, pp. 211-220, 2005.

[KP88] **Krasner, G. E. und Pope, S. T.:** "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80" in Journal of Object-Oriented Programming, 1 (3), pp. 26-49, 1988.

[KR06] **Kalofonos, D. N. und Reynolds, F. D.:** "Task-Driven End-User Programming of Smart Spaces Using Mobile Devices", Nokia Research Center, 2006.

[Kai05] **Kaisler, S. H.:** "Software paradigms", Hoboken, NJ: Wiley-Interscience, 2005.

[Kal89] **Kalb, K.:** "Memo 5: Ein merkwürdiges Verfahren, Potenzen zu berechnen (unveröffentlicht)", Universität Erlangen, 1989.

[Ker91] **Kernighan, B. W.:** "PIC-a graphics language for typesetting: user manual", Murray Hill, NJ, USA: Bell Laboratories, 1991.

[LAB03] **LABRI:** "Domain-Specific Languages - An Overview", *http://compose.labri.fr/documentation/dsl/dsl_overview.php3,* 2003, besucht am: 25.08.2005.

[Lar02] **Larman, C.:** "Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process", Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[LMS05] **Leach, P., Mealing, M. und Salz, R.:** "Request for Comments: 4122: A Universally Unique IDentifier (UUID) URN Namespace", Network Working Group, IETF, 2005.

[LR94] **Ladd, D. A. und Ramming, J. C.:** "Two Application Languages in Software Production", USENIX Symposium on Very High Level Languages, New Mexico, pp. 169-177, 1994.

[LSS+02] **Lumpe, M., Schneider, J., Schönhage, B. und Genssler, T.:** "Composition Languages", ECOOP 2002, LNCS 2548, Berlin [et al.]: Springer, pp. 107-116, 2002.

[Lex05] **Lex:** "The Lex & Yacc Page", *http://dinosaur.compilertools.net/,* 2005, besucht am: 04.10.2005.

[Loe03] **Löwy, J.:** "Programming.NET components", Sebastopol, CA, USA: O'Reilly, 2003.

[MMM+02] **Morrison, M., Morrison, J., Melrose, J., Wilson, E.:** "A Visual Code Inspection Approach to Reduce Spreadsheet Linking Errors" in Journal of End User Computing, Jul-Sep 14 (3), pp. 51-63, 2002.

[MPN+03] **McCoy, D., Pezzini, M., Natis, Y., Schulte, R., Thompson, J., Lheureux, B., Sinur, J. und Kenney, F.:** "Hype Cycle for Application Integration and Platform Middleware", Gartner Inc., 2003.

[MS98] **Mikhajlov, L. und Sekerinski, E.:** "A Study of The Fragile Base Class Problem", 12th European conference on Object Oriented Programming / ECOOP '98, Brussels, Belgium, July 20 - 24, 1998, LNCS 1445, Berlin Heidelberg: Springer, pp. 355-382, 1998.

[MW94] **Maulsby, D., Witten, I.:** "Metamouse: An Instructible Agent for Programming by Demonstration" in Allen Cypher, eds., Watch what I do: programming by demonstration. Cambridge, Mass. [u.a.]: The MIT Press, 1994.

[MWL05] **Mössenböck, H., Wöß, A. und Löberbauer, M.:** "The Compiler Generator Coco/R", *http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/,* 2005, besucht am: 04.10.2005.

[Mal93] **Malkin, G.:** "Request for Comments: 1393: Traceroute Using an IP Option", Network Working Group, IETF, 1993.

[McG02] **McGill, T.:** "User-Developed Applications: Can End Users Assess Quality?" in Journal of Organizational and End User Computing, Jul-Sep 14 (3), pp. 1-15, 2002.

[McG04] **McGill, T.:** "The Effect of End User Development on End User Success" in Journal of Organizational and End User Computing, Jan-Mar 16 (1), pp. 41-58, 2004.

[McI68] **McIlroy, M. D.:** "Mass Produced Software Components", Software Engineering: Report on a Conference by the NATO Science Committee, Garmisch, Germany, 17-11 Oct., Brüssel, Belgien, pp. 138-150, 1968.

[Mey97] **Meyer, B.:** "Object-oriented software construction", New York: Prentice Hall, 1997.

[Mic06a] **Microsoft Corporation:** ".NET Framework Class Library - IServiceProvider Interface", http://msdn.microsoft.com/library/default.asp?url=/library/en-*us/cpref/html/frlrfsystemiserviceproviderclasstopic.asp,* 2006, besucht am: 13.02.2006.

[Mic06b] **Microsoft Corporation:** ".NET Framework Class Library - IServiceContainer Interface", http://msdn.microsoft.com/library/default.asp?url=/library/en-*us/cpref/html/frlrfsystemcomponentmodeldesigniservicecontainerclasstopic.asp,* 2006, besucht am: 13.02.2006.

[Mic06c] **Microsoft Corporation:** "Visual C# Developer Center", *http://msdn.microsoft.com/vcsharp/,* 2006, besucht am: 18.07.2006.

[Mic06d] **Microsoft Corporation:** "Visual Basic Development Center", *http://msdn.microsoft.com/vbasic/,* 2006, besucht am: 29.06.2006.

[Mic06e] **Microsoft Corporation:** "Visual J# Development Center", *http://msdn.microsoft.com/vjsharp/,* 2006, besucht am: 29.06.2006.

[Mic06f] **Microsoft Corporation:** ".NET Framework Developer's Guide - Assembly Benefits", http://msdn.microsoft.com/library/default.asp?url=/library/en-*us/cpguide/html/cpconwhyuseassemblies.asp,* 2006, besucht am: 10.02.2006.

[Mic06g] **Microsoft Corporation:** "Visual C# Developer Center : Future Versions", *http://msdn.microsoft.com/vcsharp/future/,* 2006, besucht am: 10.04.2006.

[Mic06h] **Microsoft Corporation:** ".NET Framework Class Library - System.Reflection Namespace", *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemreflection.asp,* 2006, besucht am: 03.04.2006.

[Mic06i] **Microsoft Corporation:** ".NET Framework Class Library - Delegate Class", *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemDelegateClassTopic.asp,* 2006, besucht am: 28.05.2006.

[Mic06j] **Microsoft Corporation:** "Microsoft Message Queuing", *http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx,* 2006, besucht am: 23.06.2006.

[Mic06k] **Microsoft Corporation:** ".NET Framework Class Library - PropertyGrid Class", *http://msdn2.microsoft.com/en-us/library/system.windows.forms.propertygrid.aspx,* 2006, besucht am: 28.06.2006.

[Mic06l] **Microsoft Corporation:** ".NET Framework Class Library - ICustomTypeDescriptor Interface", *http://msdn2.microsoft.com/en-us/library/system.componentmodel.icustomtypedescriptor.aspx,* 2006, besucht am: 28.06.2006.

[Mic06m] **Microsoft Corporation:** ".NET Framework Class Library - System.Reflection.Emit Namespace", *http://msdn2.microsoft.com/en-us/library/system.reflection.emit(VS.80).aspx,* 2006, besucht am: 18.04.2006.

[Mic99] **Microsoft Corp.:** "Microsoft Portable Executable and Common Object File Format Specification - Revision 6.0", 1999.

[Moe51] **Moessner, A.:** "Eine Bemerkung über die Potenzen der natürlichen Zahlen" in Sitzungsberichte der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse, pp. 29, 1951.

[Mon06] **Mono Project:** "Main Page - Mono", *http://www.mono-project.com/Main_Page,* 2006, besucht am: 30.01.2006.

[Mue03] **Müller, B.:** "Netzwerke: planen, organisieren, sichern", München/Germany: Markt und Technik, 2003.

[NL97] **Nierstrasz, O. und Lumpe, M.:** "Komponenten, Komponentenframeworks und Gluing", Software Composition Group, University of Berne, Berne, Berne, 1997.

[Neu00] **Neumann, P. G.:** "Risks to the public in computers and related systems" in SIGSOFT Software Engineering Notes, 25 (3), pp. 15-23, 2000.

[Neu03] **Neumann, P. G.:** "Risks to the public in computers and related systems" in ACM SIGSOFT Software Engineering Notes, 28 (6), pp. 6-14, 2003.

[OM01] **Ostermann, K. und Mezini, M.:** "Object-Oriented Composition Untangled", Proceedings OOPSLA 2001, Tampa Bay, FL, pp. 283-299, 2001.

[Obj03a] **Object Management Group (OMG):** "Unified Modeling Language Specification, Version 1.5", *http://www.omg.org/cgi-bin/doc?formal/03-03-01,* 2003, besucht am: 17.11.2004.

[Obj03b] **Object Management Group (OMG):** "MDA Guide Version 1.0.1", *http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf,* 2003, besucht am: 13.09.2005.

[Ous97] **Ousterhout, J. K.:** "Scripting: Higher Level Programming for the 21st Century", Sun Microsystems Laboratories, 1997.

[PB96] **Pomberger, G. und Blaschek, G.:** "Software Engineering: Prototyping und objektorientierte Software-Entwicklung", München [u.a.]: Hanser, 1996.

[PB97] **Perl, J. und Bonath, K.:** "Informatik im Sport: ein Handbuch", Schorndorf: Hofmann, 1997.

[PBP+02] **Perry, M., Balachandran, M., Plata, J., Solano, P. und Thomas, P.:** "MQSeries Programming Patterns", Hursley, Hampshire, SO21 2JN United Kingdom: IBM, 2002.

[PM02] **Powell, A., Moore, J.:** "The Focus of Research in End User Computing: Where Have We Come Since the 1980s?" in Journal of Organizational and End User Computing, Jan-Mar 14 (1), pp. 3-22, 2002.

[Par72a] **Parnas, D. L.:** "A Technique for Software Module Specification with Examples" in Communications of the ACM, 15 (5), pp. 330-336, 1972.

[Par72b] **Parnas, D. L.:** "On the Criteria To Be Used in Decomposing Systems into Modules" in Communications of the ACM, 15 (12), pp. 1053-1058, 1972.

[Per02] **Perl, J.:** "Modellbildung in der Sportwissenschaft", Schorndorf: Hofmann, 2002.

[Per51] **Perron, O.:** "Beweis des Moessnerschen Satzes" in Sitzungsberichte der Bayerischen Akademie der Wissenschaften, Mathematisch-naturwissenschaftliche Klasse, pp. 31-34, 1951.

[Per81] **Perl, J.:** "Graphentheorie: Grundlagen und Anwendungen", Wiesbaden: Akademische Verlagsgesellschaft, 1981.

[Per97] **Perl, J.:** "Möglichkeiten und Probleme der computergestützten Interaktionssanalyse am Beispiel Handball" in J. Perl, ed., Sport und Informatik V, Bericht über den 5. Workshop Sport und Informatik. Köln: Sport und Buch Strauss, 1997, pp. 74-89.

[Pic05] **Pickard, M. M.:** "Old issues, new eyes", WEUSE I: Proceedings of the first workshop on End-user software engineering, New York, NY, USA: ACM Press, pp. 1-3, 2005

[Pot96] **Potel, M.:** "Model-View-Presenter. The Taligent Programming Model for C++ and Java", *ftp://www6.software.ibm.com/software/developer/library/mvp.pdf*, 1996, besucht am: 23.10.2005.

[RB05] **Ruthruff, J. R. und Burnett, M.:** "Six challenges in supporting end-user debugging", WEUSE I: Proceedings of the first workshop on End-user software engineering, New York, NY, USA: ACM Press, pp. 1-6, 2005.

[RJB99] **Rumbaugh, J., Jacobson, I. und Booch, G.:** "The unified modeling language reference manual", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 1999.

[RW82] **Rosenstengel, B. und Winand, U.:** "Petri-Netze: eine anwendungsorientierte Einführung", Braunschweig [u.a.]: Vieweg, 1982.

[Ree79] **Reenskaug, T.:** "Model - Views - Controllers", Xerox PARC, 1979.

[Rei00] **Reinhardt, J.:** "CADMOS - Component Driven Graphs als Grundlage von Softwaresystemen zur modellbasierten Systembeobachtung", Institut für Informatik, Johannes Gutenberg-Universität, Mainz, Mainz, 2000.

[Ric06] **Richter, J.:** "CLR via C#", Redmond, USA: Microsoft Press, 2006.

[Ric99] **Richter, C.:** "Designing flexible object-oriented systems with UML", Indianapolis, USA: Macmillan Technical Publishing, 1999.

[Rit06] **Ritchie, B.:** ".NET Languages", *http://www.dotnetpowered.com/languages.aspx,* 2006, besucht am: 09.04.2006.

[Rub06] **Ruby:** "Ruby Home Page", *http://www.ruby-lang.org/en/,* 2006, besucht am: 18.04.2006.

[SCK+96] **Simos, M., Creps, D., Klingler, C., Levine, L. und Allemang, D.:** "Organization Domain Modeling (ODM) Guidebook, Version 2.0", Software Technology for Adaptable, Reliable Systems, (STARS), 1996.

[SGM03] **Szyperski, C., Gruntz, D. und Murer, S.:** "Component software: beyond object-oriented programming", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 2003.

[SL05] **Spillner, A. und Linz, T.:** "Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester", Heidelberg: dpunkt-Verlag, 2005.

[SNS03] **Stutz, D., Neward, T. und Shilling, G.:** "Shared source CLI essentials", Sebastopol, CA, USA: O'Reilly, 2003.

[SVB03] **Sora, I., Verbaeten, P. und Berbers, Y.:** "A Description Language for Composable Components", Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, LNCS 2621, Berlin [et al.]: Springer, pp. 22-36, 2003.

[Sch03] **Schroeder, U.:** "Implementierung von eLearning-Szenarien nach der Theorie der kognitiven Lehre" in A. Bode, J. Desel, S. Rathmayer und M. Wessner, eds., DeLFI 2003: Tagungsband der 1. e-Learning Fachtagung Informatik. Bonn: Gesellschaft für Informatik, 2003.

[Sch98] **Schiffer, S.:** "Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten", Reading, Mass. [u.a.]: Addison-Wesley-Longman, 1998.

[Sma04] **Smaragdakis, Y.:** "A Personal Outlook on Generator Research", Domain-Specific Program Generation, LNCS 3016, Berlin Heidelberg: Springer, pp. 92-106, 2004.

[Sma99] **Smaragdakis, Y.:** "Implementing Large-Scale Object-Oriented Components", Department of Computer Sciences, University of Texas, Austin, TX, USA, Austin, TX, USA, 1999.

[Sof05] **Software Engineering Institute - Carnegie Mellon University:** "A Framework for Software Product Line Practice, Version 4.2", *http://www.sei.cmu.edu/productlines/framework.html,* 2005, besucht am: 12.09.2005.

[Som01] **Sommerville, I.:** "Software Engineering", München: Pearson Studium, 2001.

[Spa00] **Sparling, M.:** "Lessons learned through six years of component-based development" in Communications of the ACM, 43 (10), pp. 47-53, 2000.

[Sta06] **Stagecast Software Inc.:** "About Stagecast Creator", *http://www.stagecast.com/creator.html,* 2006, besucht am: 18.01.2006.

[Sug01] **Sugiura, A.:** "Web Browsing by Example" in H. Lieberman, eds., Your wish is my command: programming by example. San Francisco [u.a.]: Morgan Kaufmann, 2001, pp. 61-86.

[Sun04] **Sun Microsystems:** "API Dokumentation zu java.lang.ClassLoader", *http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html,* 2004, besucht am: 14.03.2006.

[Sun06] **Sun Microsystems:** "Enterprise Java Beans Specifications", *http://java.sun.com/products/ejb/docs.html,* 2006, besucht am: 22.01.2006.

[Sut05] **Sutcliffe, A.:** "Evaluating the costs and benefits of end-user development", WEUSE I: Proceedings of the first workshop on End-user software engineering, New York, NY, USA: ACM Press, pp. 1-4, 2005.

[Syb06] **Sybase:** "Business Process Modeling, Application Design, J2EE Development - Sybase PowerDesigner", *http://www.sybase.com/products/developmentintegration/powerdesigner,* 2006, besucht am: 18.03.2006.

[TFH05] **Thomas, D., Fowler, C. und Hunt, A.:** "Programming Ruby: the pragmatic programmers' guide", Raleigh, NC: The Pragmatic Bookshelf, 2005.

[Ter05] **Terry, P.:** "Compiling with C# and Java", Harlow: Pearson/Addison-Wesley, 2005.

[Tho04] **Thomas, D.:** "Message Oriented Programming" in Journal of Object Technology, May-June 3 (5), pp. 7-12, 2004.

[Tol01] **Tolvannen, J.:** "Domänenspezifische Modellierungssprachen für Produktfamilien" in OBJEKTspektrum, (5), pp. 17-22, 2001.

[Tra03] **TransAlta:** "TransAlta provides second quarter update - Press Release", *http://www.transalta.com/WEBSITE2001/TAWEBSITE.NSF/AllDoc/03CA0B770CF02D8A87256D3A00667FA2?OpenDocument,* 2003, besucht am: 11.01.2006.

[VP03] **Vantroys, T. und Peter, Y.:** "COW, a Flexible Platform for the Enactment of Learning Scenarios", CRIWG 2003, LNCS 2806, Berlin Heidelberg: Springer, pp. 168-182, 2003.

[Van05] **Vanderseypen, F.:** "The Netron Project", *http://www.netronproject.com/,* 2005, besucht am: 12.10.2005.

[Voe03] **Voelter, M.:** "A Catalog of Patterns for Program Generation", Proceedings of the EuroPLoP 2003 conference, Irsee, Germany, 2003

[W3C03] **W3C XML Protocol Working Group:** "SOAP Version 1.2 Usage Scenarios", *http://www.w3.org/TR/2003/NOTE-xmlp-scenarios-20030730,* 2003, besucht am: 30.05.06.

[WJ04] **Wulf, V. und Jarke, M.:** "The economics of end-user development" in Communications of the ACM, 47 (9), pp. 41-42, 2004.

[WK94] **Willliams, S., Kindel, C.:** "The Component Object Model: A Technical Overview", Microsoft Corp., 1994.

[Wag02] **Wagner, C.:** "End Users as Expert System Developers?" in Tonya Barrier, eds., Human computer interaction development and management. Hershey: IRM Press, 2002.

[Wal82] **Wall, D. W.:** "Messages as active agents", POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA: ACM Press, pp. 34-39, 1982.

[Wat86] **Waterman, D. A.:** "A guide to expert systems", Upper Sadle River N.J., Boston, [u.a.]: Addison-Wesley, 1986.

[Weg90] **Wegner, P.:** "Concepts and paradigms of object-oriented programming" in SIGPLAN OOPS Messenger, 1 (1), pp. 7-87, 1990.

[Wei06a] **Weifen, L.:** "SourceForge.Net: DockPanel Suite", *http://sourceforge.net/projects/dockpanelsuite/,* 2006, besucht am: 02.03.2006.

[Wei06b] **Weirich, J.:** "10 Things Every Java Programmer Should Know About Ruby", *http://onestepback.org/articles/10things/,* 2006, besucht am: 24.04.2006.

[Whi96] **Whitey, J.:** "Investment Analysis of Software Assets for Product Lines", Software Engineering Institute, Carnegie Mellon University, Pittsburgh USA, 1996.

[Wor01] **World Wide Web Consortium (W3C):** "XML Base - W3C Recommendation 27 J une 2001", *http://www.w3.org/TR/2001/REC-xmlbase-20010627/,* 2001, besucht am: 12.05.2006.

[ZCZ97] **Zhao, J., Chen, J. und Zheng, G.:** "Message conversion and a new type system for OO model" in ACM SIGPLAN Notices, 32 (9), pp. 61-67, 1997.

[Zwi05] **Zwintzscher, O.:** "Software-Komponenten im Überblick: Einführung, Klassifizierung Vergleich von JavaBeans, EJB, COM+, Net, CORBA, UML 2", Bochum: W3L-Verl., 2005.

[db406] **db4objects, I.:** "db4o :: Native Java &.NET Object Database :: Open Source", *http://www.db4o.com/,* 2006, besucht am: 15.06.2006.

Abkürzungsverzeichnis

API	Application Programming Interface
AOP	Aspekt Orientierte Programmierung
AST	Abstract Syntax Tree
BO	Benutzungsoberfläche
CASE	Computer Aided Software Engineering
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CodeDOM	Code Document Object Model
COM	Component Object Model
CTS	Common Type System
DLL	Dynamic Link Library
DSEL	Domain Specific Embedded Language
DSL	Domain Specific Language
ECMA	European Computer Manufacturers Association
EJB	Enterprise Java Beans
EUC	End User Computing
EUD	End User Development
EUP	End User Programming
FIFO	First-In-First-First-Out
GAC	Global Assembly Cache
GPL	General Purpose Language
GUI	Graphical User Interface
GUID	Global Unique Identifier
HTTP	Hypertext Transfer Protocol
IC	Integrated Circuit
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IDL	Interface Description Language
ISO	International Organization for Standardization
JIT	Just In Time
MDA	Model Driven Architecture
MDI	Multi Document Interface
MOP	Message Oriented Programming
MSIL	Microsoft Intermediate Language

MVC	Model View Controller
MVC'	Mode-View-Connector
RPC	Remote Procedure Call
SOA	Service Oriented Architecture
SSCLI	Shared Source CLI
UML	Unified Modelling Language
VB.NET	Visual Basic for .NET
VBA	Visual Basic for Applications
VES	Virtual Execution System
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
XAML	XML Application Markup Language
XML	eXtensible Markup Language
XSL	eXtensible Styleheet Language
XSLT	eXtensible Styleheet Language for Transformations
ZE-Graph	Zustands-Ereignis-Graph