

Horst Lichter

**Entwicklung und
Umsetzung von
Architekturprototypen für
Anwendungssoftware**

Von der Fakultät Informatik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung.

Vorgelegt von Horst Lichter aus Trier.

Hauptberichter:	Prof. Dr. Jochen Ludewig
Mitberichter:	Prof. Dr. Rul Gunzenhäuser
Tag der Einreichung:	2. März 1993
Tag der mündlichen Prüfung:	29. April 1993

meinen Eltern gewidmet

für ihre Unterstützung und ihr Vertrauen

Viele haben – direkt oder indirekt – dazu beigetragen, daß diese Arbeit durchgeführt und abgeschlossen werden konnte. An erster Stelle ist Herr Professor Jochen Ludewig zu nennen. Er hat mich fachlich und menschlich betreut, zuerst während der gemeinsamen Zeit an der ETH Zürich, in der die grundlegenden Ideen für diese Arbeit entstanden, und anschließend an der Universität Stuttgart. Er hat mir die Freiheiten gelassen, diese Arbeit durchzuführen, und in vielen Gesprächen Anregungen und kritische Anmerkungen zur Arbeit gegeben. Bei Herrn Professor Rul Gunzenhäuser bedanke ich mich, daß er den Mitbericht übernommen hat.

Bei Angela, Marcus, Kurt, Max und Jürgen möchte ich mich für die angenehme Arbeitsatmosphäre in der Abteilung Software Engineering bedanken. Speziell danke ich Angela und Kurt dafür, daß sie im Projekt vis-A-vis mitgearbeitet haben. André sei gedankt, weil er half, die Motif-Benutzungsoberfläche einiger Werkzeuge zu implementieren.

Nicht zuletzt möchte ich meiner Frau Gabi dafür danken, daß sie mich unterstützt hat und mir – immer dann, wenn es notwendig war – Mut gemacht hat, die Arbeit abzuschließen.

„Alles Wissen und alle Vermehrung unseres Wissens endet nicht mit einem Schlußpunkt, sondern mit einem Fragezeichen.“

Ein Plus an Wissen bedeutet ein Plus an Fragestellungen, und jede von ihnen wird immer wieder von neuen Fragestellungen abgelöst.“

HERMANN HESSE (1877-1962)

Kurzfassung

Prototyping wird in den letzten Jahren zunehmend eingesetzt, damit Software-Projekte planbarer und auch erfolgreicher werden. In dieser Arbeit wird ein Ansatz für das Prototyping präsentiert, bei dem die Software-Architektur, die die Qualität der Software entscheidend beeinflusst, mit einem Prototyp untersucht wird.

Zuerst wird herausgearbeitet, wie *Prototyping* mit der Terminologie der allgemeinen *Modelltheorie* beschrieben werden kann. Es wird ein Software-Modellierungsansatz vorgestellt, der auf den Dokumenten basiert, die bei der Software-Entwicklung erstellt werden. Die Betrachtungen zur Modellierung von Software führen zu einer Klassifikation von Software-Modellierungsansätzen. Diese Klassifikation definiert die Software-Architekturmodellierung als einen speziellen Ansatz der Software-Modellierung. Sie führt weiterhin das *Software-Architektur-Prototyping* als Spezialisierung der Software-Architekturmodellierung ein. Charakteristisch für das Software-Architektur-Prototyping ist, daß dabei ein ausführbares Modell der Programmarchitektur entsteht – der Architekturprototyp.

Auf der Basis dieser konzeptionellen Ergebnisse wird eine Vorgehensweise für das Software-Architektur-Prototyping vorgestellt, die PDSC (**P**rogram **D**evelopment by **S**teps**w**ise **C**ompletion) genannt wird. Sie sieht vor, daß der konstruierte Architekturprototyp schrittweise in das Zielprogramm überführt wird. Es wird diskutiert, warum dies sinnvoll ist und welche Vorteile diese Vorgehensweise hat. Weiterhin wird beschrieben, welche Aktivitäten bei PDSC durchgeführt werden müssen, wie der Übergang im einzelnen vonstatten geht, welche Sprachen dazu notwendig sind und welche Werkzeuge diese Vorgehensweise operational unterstützen können.

Im letzten Teil der Arbeit wird gezeigt, wie die vorgestellten Ideen zum Software-Architektur-Prototyping konstruktiv umgesetzt werden können. Dies wird am Beispiel der *objektorientierten* Software-Entwicklung getan. Dazu werden zwei Sprachen vorgestellt: *ADL*, um Modelle objektorientierter Architekturen zu erstellen, und *ProST*, um entsprechende Architekturprototypen zu realisieren. Als Zielsprache wird *Eiffel* gewählt. Abschließend werden Werkzeuge präsentiert, die das Prototyping objektorientierter

Architekturen gemäß PDSC unterstützen. Diese Werkzeuge sind in der Werkstatt *ProWork* zusammengefaßt.

Abstract

Prototyping has, in recent years, become an approach to improve the plannability – and thus, the success – of software projects. In this thesis, a special integrated prototyping approach is introduced. This approach called *software architecture prototyping* consists of a method, corresponding languages, and a set of tools. Since the architecture substantially influences the overall software quality, it is the central aspect examined by a prototype. To evaluate alternative architectures a so-called *architecture prototype* is built.

In the first part, a terminology for prototyping is presented and the relationship between prototyping and Stachiowiak's general model theory is shown. According to the general model theory, a prototype is defined to be a special computer model. Based on the results obtained, the modeling of software is discussed and a taxonomy of software modeling approaches is introduced. This taxonomy classifies software-architecture modeling as a specialization of software modeling in general. Subsequently, software architecture prototyping is defined as a special approach to software architecture modeling. In the process of software architecture prototyping, an executable model of the program architecture is built.

In the second part, a method for software architecture prototyping is introduced, called PDSC (**P**rogram **D**evelopment by **S**tepwise **C**ompletion). Using PDSC, an architecture prototype is completed stepwise and transformed into the final program. The main advantages of this method are discussed and the activities which have to be performed in the process of building and transforming the architecture prototype are described. Furthermore, requirements for languages used within PDSC are stated.

To illustrate the approach introduced in this thesis, the last part describes how architecture prototypes may be build and completed based on a special software design strategy. For this purpose, the *object-oriented* development strategy is chosen. Two languages are introduced, *ADL* to model object-oriented architectures, and *ProST* to implement the architecture prototype. To implement the final program, *Eiffel* was chosen. A prototyping environment is explained which consists of tools supporting the construction and completion of object-oriented architecture prototypes. The environment called *ProWork* offers tools like a graphical editor for ADL, a prototype

configurator, and generators to transform an ADL description into a corresponding ProST or Eiffel description.

Inhalt

Kurzfassung	11
Abstract	12
<hr/>	
1 Zielsetzung der Arbeit	13
1.1 Einleitung	13
1.2 Prototyping in der industriellen Software-Entwicklung	14
1.3 Die untersuchten Fragestellungen	15
1.4 Aufbau der Arbeit	17
<hr/>	
2 Grundlagen	19
2.1 Begriffsklärung	19
2.1.1 Konzept, Methode, Sprache, Werkzeug	19
2.1.2 Software, Programm, Anwendungssoftware	20
2.1.3 Zielsoftware, Zielprogramm, Zielsprache	21
2.1.4 Prototyp, Prototyping, Prototyping-Sprache	21
2.2 Die Modellbildung	22
2.2.1 Modell, Computermodell, Modellieren	22
2.2.2 Die Beziehung zwischen Modell und Original	24
2.2.3 Simulation	26
2.3 Software-Prototyping	26
2.3.1 Klassifizierung von Prototyping	28
2.3.2 Modellierung und Prototyping	30
2.3.3 Der allgemeine Entwicklungszyklus beim Prototyping	32
<hr/>	
3 Software-Modellierung und Software-Architekturen	35
3.1 Prozeßmodelle und Software-Modelle	35
3.2 Ebenen der Modellierung	36
3.2.1 Die Schemaebene	36
3.2.2 Die Modellebene	37
3.2.3 Die Ausprägungsebene	37

3.2.4	Ein einfaches Beispiel	38
3.3	Die dokumentenbasierte Modellierung der Software	39
3.3.1	Software-Entwicklung ist Dokumentenentwicklung	39
3.3.2	Grundlegende Beziehungen zwischen Dokumenten	40
3.4	Die Entwicklungsgeschichte der Dokumente	42
3.4.1	Einführung	42
3.4.2	Die Revisionenbildung	43
3.4.3	Die Variantenbildung	44
3.4.4	Die Versionenbildung im Detail	45
3.5	Ein dokumentenbasiertes Software-Schema	46
3.5.1	Eine Notation für Software-Schemata	47
3.5.2	Das DS-Schema	48
3.6	Die Architektur der Dokumente	51
3.6.1	Ein allgemeiner Architekturbegriff	51
3.6.2	Software-Architektur und Software-Architekturdokumente	52
3.6.3	Die Software-Entwicklung und der Bau eines Gebäudes	52
3.6.4	Architekturen bei der Software-Entwicklung	53
3.7	Eine Klassifikation für Software-Modellierungsansätze	55

4	Software-Architektur-Prototyping - Prinzip und Vorgehensweise	57
4.1	Warum Software-Architektur-Prototyping?	57
4.1.1	Zusammenhang zwischen Programmarchitekturentwurf und Implementierung	57
4.1.2	Das Prinzip des Software-Architektur-Prototyping	58
4.1.3	Charakterisierung des Software-Architektur-Prototyping	59
4.1.4	Der Übergang vom Architekturmodell zum Zielsystem	60
4.2	PDSC – Der schrittweise Übergang vom Architekturprototyp zum Zielprogramm	61
4.2.1	Unvollständigkeiten bei der Software-Entwicklung	61
4.2.2	Die schrittweise Vervollständigung des Architekturprototyps	62
4.2.3	Tätigkeiten und Rollen bei PDSC	64
4.3	Übergang von Prototyp- zur Zielsprachenimplementierung	68
4.3.1	Wann geschieht der Übergang?	68
4.3.2	Die Reihenfolge beim Übergang	69
4.4	Sprachen für PDSC	70
4.4.1	Anforderungen an PDSC-Sprachen	70

4.4.2	Beziehungen zwischen Prototyping- und Zielsprache	71
4.5	Die Entwicklungsgeschichte der Dokumente bei PDSC	72
4.6	Konzeption einer Werkstatt für PDSC	74
4.6.1	Die Datenhaltung	74
4.6.2	Die Werkzeuge einer Werkstatt für PDSC	75
4.7	Bewertung von PDSC	79
<hr/>		
5	Ein Software-Schema für das Architektur-Prototyping	81
5.1	Ansätze zur Definition von Software-Schemata	81
5.1.1	Der Etikettierungsansatz	82
5.1.2	Der Spezialisierungsansatz	83
5.1.3	Bewertung der beiden Ansätze	83
5.2	Programmmentwurfsmethode und Programmschema	84
5.3	Anforderungen an ein Software-Schema für PDSC	85
5.4	Ein Software-Schema für PDSC	85
5.4.1	Die Bausteinararten des Schemas	87
5.4.2	Die Beziehungsarten des Schemas	87
5.4.3	Bewertung des Schemas	89
5.5	Bisher erzielte Ergebnisse	89
<hr/>		
6	Prototyping objektorientiert entworfener Architekturen	93
6.1	Architektur-Prototyping im Überblick	93
6.1.1	Konzepte, Methoden, Sprachen und Werkzeuge	93
6.1.2	Tätigkeiten	95
6.2	Ein Software-Schema für objektorientierte Programmarchitekturen	96
6.2.1	Vorbemerkungen	97
6.2.2	Warum eine objektorientierte Erweiterung?	97
6.2.3	Ein Schema für objektorientierte Programmarchitekturen	98
6.3	Das OOS-Schema	100
6.3.1	Die Bausteinararten des OOS-Schemas	100
6.3.2	Die Beziehungsarten des OOS-Schemas	103
6.3.3	Bewertung des Schemas	106
6.4	Sprachen für das Architektur-Prototyping	107
6.4.1	Die Software-Modellierungssprache	108
6.4.2	Die Prototyping-Sprache	108
6.4.3	Die Zielsprache	108

6.5	Konzeption der Sprachen ADL und ProST	109
6.5.1	Beziehungen zwischen den Sprachen	109
6.5.2	Entwurfsentscheidungen für ADL und ProST	110
6.5.3	Zur Notation der Syntaxdiagramme	111
6.5.4	Bezeichner in ADL und ProST	111
6.6	Die Architekturbeschreibungssprache ADL	112
6.6.1	Die grafischen Symbole für Dokumente und Beziehungen	113
6.6.2	Zusätzliche Angaben	113
6.6.3	Klassen und Eigenschaften	114
6.7	ProST: eine Smalltalk-80 basierte Prototyping-Sprache	118
6.7.1	Vorbemerkungen	119
6.7.2	Klassen in ProST	119
6.7.3	Routinen in ProST	121
<hr/>		
7	Die Prototyping-Werkstatt ProWork	123
7.1	Werkzeug und Werkstatt	123
7.2	Die Werkstatt im Überblick	124
7.2.1	Die angebotenen Werkzeuge	124
7.2.2	Der Gebrauch der Werkzeuge	125
7.2.3	Anmerkungen zur Realisierung	126
7.3	Der Architektureditor für ADL	127
7.3.1	Das „Application Framework vis-A-vis“	128
7.3.2	Der Architektureditor	129
7.4	Die ProST-Sprachwerkzeuge	130
7.4.1	Die Integration von ProST in die Smalltalk-80 Programmierumgebung	130
7.4.2	ProST-Browser und -Compiler	132
7.5	Die Datenhaltung	133
7.5.1	GemStone - Ein Überblick	133
7.5.2	Die ProWork-Datenbank	134
7.5.3	Das Datenbankschema	135
7.6	Die hybride Ausführung von ProST und Eiffel	136
7.6.1	Das gewählte Konzept	136
7.6.2	Die Anbindung der Sprachen – Ein technischer Überblick	136
7.6.3	Die Realisierung der Schnittstelle	139
7.6.4	Die Werkzeugunterstützung	140
7.6.5	Die Anbindung von Eiffel an ProST - Ein Beispiel	142

7.7 Sonstige Werkzeuge	144
7.7.1 Der Prototypkonfigurator	144
7.7.2 Der Datenbankmanager	146
7.7.3 Das Ausführungswerkzeug	147
<hr/>	
8 Beispiel, Vergleich und Bewertung	149
8.1 Ein Beispiel	149
8.1.1 Die Problemstellung	150
8.1.2 Das Architekturmodell	151
8.1.2 Erprobung von Versionen	154
8.1.3 Der Übergang zum Zielprogramm	158
8.2 Vergleich mit verwandten Arbeiten	160
8.2.1 TOPOS	160
8.2.2 PROTOS	162
8.2.3 CAPS	164
8.2.4 ProLab	165
8.3 Zusammenfassung	167
8.4 Bewertung des Ansatzes	168
8.4.1 Stärken des Ansatzes	169
8.4.2 Schwächen und nicht gelöste Probleme	170
<hr/>	
Literatur	173

Kapitel 1

Zielsetzung der Arbeit

1.1 Einleitung

Aufgrund der Probleme bei der Entwicklung von Software nach traditionellen Prozeßmodellen – wie etwa dem in Boehm (1976) beschriebenen Wasserfallmodell – und der daran geäußerten Kritik wird *Prototyping* in den letzten Jahren zunehmend eingesetzt, damit Software-Projekte planbarer und auch erfolgreicher werden. Mit Prototyping wird sowohl die Qualität der Software als auch die Entwicklungseffizienz verbessert. Diese Ziele werden erreicht, weil mit Prototypen die Aspekte der Software frühzeitig untersucht werden können, die besonders risikoreich sind und damit eine Gefahr für das gesamte Projekt darstellen. Die Konstruktion von Prototypen kann in Prozeßmodelle integriert werden, wie es z.B. beim Spiralenmodell, das in Boehm (1988) beschrieben wird, geschehen ist.

Prototyping ist seit einigen Jahren ein Arbeitsgebiet im Software-Engineering. Es werden spezielle Tagungen dazu abgehalten, und es gibt eine Vielzahl von Veröffentlichungen zum Thema Prototyping. Die Literatur kann folgendermaßen klassifiziert werden:

- Veröffentlichungen, die Prototyping umfassend diskutieren und dabei Konzepte, Methoden, Sprachen und Werkzeuge berücksichtigen. Dazu zählen z.B. Lanz (1986), Budde (1992c) oder Bischofberger (1992).
- Veröffentlichungen, die beschreiben, inwieweit einzelne Programmiersprachen für das Prototyping geeignet sind. Solche Arbeiten sind z.B. Duncan (1982), Diederich (1987) und Doberkat (1989).
- Erfahrungsberichte, die darstellen, wie Prototyping in konkreten Projekten eingesetzt wurde und welche Ergebnisse erzielt wurden. Dazu zählen z.B. Boehm (1984), Gordon (1991) oder Kieback et al. (1992).
- Veröffentlichungen, die einen speziellen Ansatz für das Prototyping und die dazu geeigneten Sprachen und Werkzeuge vorstellen. Zu dieser Gruppe gehören z.B. die Arbeit von Hallmann (1988), der Prototyping

für transaktionsorientierte Systeme betrachtet, und die Arbeit von Luqi (1988b), die Prototyping für Realzeit-Anwendungen untersucht.

Die vorliegende Arbeit kann in die letztgenannte Gruppe eingeordnet werden. Sie stellt einen Ansatz vor, bei dem die Architektur der Software der zentrale Aspekt ist, der mit einem Prototyp untersucht wird.

Im folgenden wird beschrieben, warum es sinnvoll ist, die Architektur der Software mit einem speziellen Prototyp, der im weiteren als *Architekturprototyp* bezeichnet wird, zu untersuchen. Dazu werden zuerst die Erfahrungen aus der industriellen Software-Entwicklung präsentiert, die mit Prototyping gemacht wurden. Anschließend werden Fragestellungen formuliert, die im Zusammenhang mit der Konstruktion von Architekturprototypen gestellt werden müssen, und es wird ein Überblick über den Aufbau der Arbeit gegeben.

1.2 Prototyping in der industriellen Software-Entwicklung

Bei industriellen Software-Projekten werden Prototypen besonders dann konstruiert, wenn es sich um neue Anwendungsgebiete handelt, auf denen bisher nur wenige oder überhaupt keine Erfahrungen gemacht wurden. Diese Projekte sind demnach mit besonderen Risiken behaftet, die mit Prototyping gemildert werden sollen.

Die von Kieback et al. (1992) veröffentlichte Studie über industrielle Software-Projekte, bei denen Prototyping explizit bei der Projektdurchführung eingesetzt wurde, kommt bei der Analyse der Projekterfahrungen zu folgenden Ergebnissen:

- Prototyping wird positiv bewertet, weil nach Aussagen der Entwickler die Problemstellung besser verstanden wird und die Anforderungen an die Software einfacher und sicherer ermittelt werden können.
- Prototypen werden meistens so konzipiert, daß der dabei geschriebene Code nicht weggeworfen, sondern verwendet wird, um das Zielsystem zu realisieren. Dies wird damit begründet, daß die Konstruktion des Prototyps meist kein eigenes abgeschlossenes Projekt ist und der in den Prototyp investierte Aufwand sich so weit als möglich produktiv auszahlen muß. Deshalb werden Prototyp und Zielsystem häufig in derselben Programmiersprache implementiert.

- Prototypen werden immer nur dann erstellt, wenn geeignete Werkzeuge oder sogar spezielle Entwicklungsumgebungen vorhanden sind, die die Konstruktion der Prototypen unterstützen. Solche Werkzeuge existieren heute, um Prototypen von Benutzungsschnittstellen zu entwickeln, sowie im Bereich der kommerziellen Datenbankanwendungen.
- Prototypen werden immer dann eingesetzt, wenn Lösungen für realisierungstechnische Probleme untersucht werden müssen.

Neben diesen durchweg positiven Erfahrungen wurden aber auch Mängel und Schwächen festgestellt:

- Prototyping führt häufig dazu, daß die Dokumentation vernachlässigt wird, da der Prototyp als Dokumentationsersatz angesehen wird. Die fehlende Dokumentation führt jedoch immer dann zu Problemen, wenn Teile eines Prototyps als Bausteine für das Zielsystem verwendet werden.
- Erfolgreiches Prototyping muß auf den Ergebnissen einer Analysephase basieren, die die wesentlichen Anforderungen feststellt und eine einheitliche Terminologie für die an der Entwicklung beteiligten Personengruppen schafft. Ist dies nicht der Fall, kann das Verhalten eines ersten Prototyps soweit von den Vorstellungen entfernt sein, daß er nicht nützlich für die Entwicklung der Software ist.

Abschließend wird in der Studie festgestellt, daß der Einsatz von Prototyping immer dann erfolgreich ist, wenn die organisatorischen Rahmenbedingungen dafür geeignet sind und alle an der Entwicklung beteiligten Personen die Stärken, aber auch die Grenzen dieser Vorgehensweise kennen.

1.3 Die untersuchten Fragestellungen

Prototyping hat sich bewährt, um einerseits zusammen mit den Anwendern die Anforderungen an die Benutzungsschnittstelle und an die Funktionalität zu ermitteln und um andererseits Lösungsalternativen für technische Probleme zu bewerten. Damit werden die Voraussetzungen verbessert, daß die am Ende der Entwicklung ausgelieferte Software dem entspricht, was der Auftraggeber und die Anwender davon erwarten.

In der Praxis wird, wie im vorangegangenen Abschnitt festgestellt wurde, aus wirtschaftlichen Gründen häufig der Prototyp, oder wenigstens Teile davon, als Basis für die Entwicklung des Zielsystems verwendet. Da Prototypen aber schnell erstellt werden müssen, können sie nicht mit

derselben Sorgfalt entworfen und codiert werden, wie es für das Zielsystem notwendig ist. An einen Prototyp werden nicht die Qualitätsanforderungen gestellt, die für das Zielsystem gefordert werden. Wird der Prototyp aber zu einem Teil des Zielsystems weiterentwickelt, besteht die Gefahr, daß die beim Prototyp noch akzeptierte mangelnde Qualität zu einer Qualität beim Zielsystem führt, die nicht mehr akzeptabel ist.

Das Resultat ist dann, wie ein Beispiel in Kieback et al. (1992) zeigt, eine Software mit schlechter Architektur, die nur schwer gewartet werden kann. Eine kosten- und zeitintensive Reimplementierung ist die notwendige Konsequenz.

Software, die eingesetzt wird, wird – wie schon Lehman (1980) feststellt – während ihrer Lebensdauer modifiziert, um sie an die sich ändernden Einsatzanforderungen anzupassen. Das gilt sowohl für das Programm als auch für die erstellte Dokumentation zum Programm. Aus diesem Grund existiert für jede Software die Anforderung, daß sie modifizierbar und erweiterbar – also wartbar – sein muß. Diese Anforderung gilt jedoch nicht in dem Maße für einen Prototyp.

Die Wartbarkeit der Software ist abhängig von der Güte ihrer Architektur, von der Qualität der erstellten Dokumentation und vom verwendeten Programmierstil. Die Architektur der Software muß so beschaffen sein, daß Modifikationen möglichst lokal durchgeführt und Erweiterungen einfach vorgenommen werden können. Dies ist umso wichtiger, da die Architektur während der Lebensdauer der Software stabil bleibt. Änderungen der Architektur sind unmöglich.

Der Entwurf der Architektur ist aus diesen Gründen für die Qualität der Software von großer Bedeutung. Da es jedoch nicht möglich ist, auf Anhieb eine Software-Architektur zu finden, die die genannten Qualitäten besitzt, ist es sinnvoll, die Software-Architektur als den zentralen Aspekt zu betrachten, der mit einem Prototyp untersucht wird.

Mit einem ausführbaren Architekturprototyp können bereits sehr früh im Software-Entwicklungsprozeß Informationen und Wissen über die Architektur erhalten werden. Wurde mithilfe dieses Prototyps eine „geeignete“ Architektur gefunden, dient diese als Vorbild bei die Entwicklung des Zielsystems.

Basierend auf diesem hier nur skizzierten Ansatz lassen sich nun Fragen formulieren, die im Zusammenhang mit der Konstruktion eines Architekturprototyps relevant sind:

- Wie kann Software und speziell die Software-Architektur modelliert werden?
- Welche Ziele sollen mit Architektur-Prototyping erreicht und welche Tätigkeiten müssen durchgeführt werden?
- Wie müssen ein Prozeßmodell und eine Methode aussehen, damit die Ergebnisse des Architektur-Prototyping in die Entwicklung des Zielsystems einfließen können?
- Welche Werkzeuge können das Architektur-Prototyping unterstützen?

Die vorliegende Arbeit diskutiert diese Fragestellungen und präsentiert einen Ansatz, um Architekturprototypen zu erstellen.

1.4 Aufbau der Arbeit

Die Arbeit besteht – neben dieser Einführung und einer abschließenden Betrachtung – aus sechs Kapitel. Diese werden in drei inhaltlich zusammengehörende Teile gegliedert:

Im ersten Teil, der aus Kapitel 2 und 3 besteht, werden die Grundlagen geschaffen, auf denen der vorgestellte Ansatz zum Software-Architektur-Prototyping basiert.

- In Kapitel 2 werden die Begriffe eingeführt, die für das Verständnis der Arbeit notwendig sind. Weiterhin wird herausgearbeitet, wie Prototyping und Modellbildung zusammenhängen.
- In Kapitel 3 wird die Modellierung der Software diskutiert und eine dokumentenbasierte Software-Modellierung vorgestellt. Darauf aufbauend wird ein Architekturbegriff für Dokumente eingeführt und Software-Architektur-Prototyping als eine spezielle Art der Software-Modellierung definiert.

Im zweiten Teil werden die zentralen Ideen zum Software-Architektur-Prototyping formuliert. Er besteht aus Kapitel 4 und 5.

- In Kapitel 4 wird beschrieben, was Software-Architektur-Prototyping ist und welche Ziele damit erreicht werden sollen. Es wird eine Vorgehensweise für das Software-Architektur-Prototyping vorgestellt, bei der der erstellte Architekturprototyp schrittweise in das Zielformat überführt wird.

- In Kapitel 5 wird ein Software-Schema definiert, das auf dem in Kapitel 3 vorgestellten Modellierungsansatz basiert und auf die beschriebene Vorgehensweise für das Software-Architektur-Prototyping angepaßt ist.

Der dritte Teil der Arbeit besteht aus Kapitel 6 und 7. In diesem Teil wird der beschriebene allgemeine Ansatz für das Architektur-Prototyping anhand einer konkreten Software-Entwicklungsstrategie umgesetzt. Als Beispiel dient die objektorientierte Software-Konstruktion.

- In Kapitel 6 wird ein Software-Schema entwickelt, mit dem objektorientierte Architekturen modelliert werden können. Es werden Sprachen erläutert, die geeignet sind, um auf der Basis dieses Schemas ein Architekturmodell zu formulieren, den Architekturprototyp zu erstellen und das Zielprogramm zu realisieren.
- In Kapitel 7 werden Werkzeuge präsentiert, die in einer Prototyping-Werkstatt zusammengefaßt sind. Diese Werkzeuge sind speziell für das Architektur-Prototyping objektorientierter Anwendungssoftware entwickelt, bei dem die in Kapitel 6 beschriebenen Sprachen eingesetzt werden.

Im letzten Kapitel wird ein Beispiel gezeigt. Der in der Arbeit vorgestellte Ansatz wird bewertet und mit ähnlichen Arbeiten verglichen. Abschließend werden Fragen und Probleme formuliert, die zwar identifiziert, aber im Rahmen dieser Arbeit nicht gelöst wurden.

Kapitel 2

Grundlagen

In diesem Kapitel werden die begrifflichen und methodischen Grundlagen eingeführt. Es werden zuerst die zentralen Begriffe definiert, die für das Verständnis der Arbeit notwendig sind. Anschließend wird die Modellbildung und das Prototyping näher betrachtet und gezeigt, wie Prototyping mit Modellierung im allgemeinen zusammenhängt.

2.1 Begriffsklärung

Die Begriffe sind weitgehend so definiert, daß sie sich auf die entsprechenden Definitionen aus der Norm IEEE (1990) abstützen. Logisch zusammengehörende Begriffe sind gruppiert. Die Definitionen werden durch Beispiele erläutert, damit sie anschaulicher sind.

2.1.1 Konzept, Methode, Sprache, Werkzeug

Ein *Konzept* ist eine Idee oder eine Grundvorstellung, nach der etwas getan werden soll. Ein Konzept in diesem Sinn ist beispielsweise die Idee, den Aufbau von Organisationen durch Entitäten und Beziehungen zu beschreiben.

Eine *Methode* ist eine Sammlung von Regeln, Verfahren und Verhaltensweisen, die geeignet sind, um eine Aufgabe planmäßig durchzuführen. Ein Beispiel für eine Methode ist der Programmentwurf, wie er in Jackson (1983) beschrieben wird.

Unter einer *Sprache* wird in dieser Arbeit immer eine formale Sprache verstanden. Eine Sprache wird nach Ludewig (1988) durch ihre Syntax und Semantik definiert. Die Syntax definiert die Menge aller zulässigen Aussagen, die mit der Sprache formuliert werden können. Die Semantik definiert die den syntaktisch korrekten Aussagen zugeordnete Bedeutung. Sprachen in diesem Sinn sind beispielsweise Programmiersprachen.

Werkzeuge sind Hilfsmittel, die benötigt werden, um eine Tätigkeit auszuführen. Unter einem Werkzeug wird in dieser Arbeit immer ein Programm verstanden, das verwendet wird, um Software zu konstruieren oder zu warten. Ein Werkzeug ist z.B. ein Editor, ein Übersetzer oder auch ein Dateivergleicher.

Konzepte, Methoden, Sprachen und Werkzeuge sind eng miteinander verbunden. In Ludewig et al. (1985a) wird anschaulich durch ein „Systemdreieck“ dargestellt, daß Konzepte, Methoden, Sprachen und Werkzeuge aufeinander abgestimmt sein müssen.

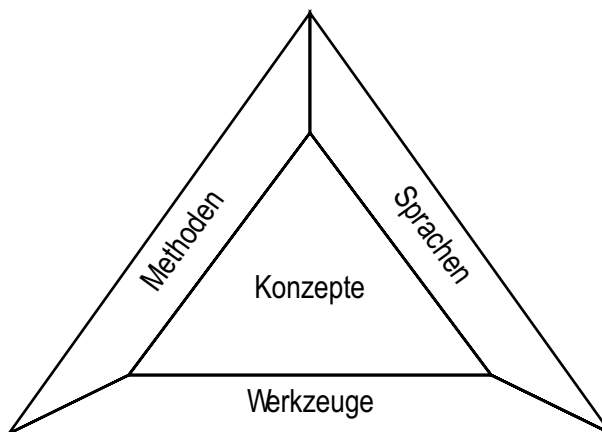


Abb. 2.1: Das Systemdreieck

Die Konzepte bilden den zentralen Kern, der die Grundlage für die Sprachen, Methoden und Werkzeuge eines Systems ist. Die benutzten Sprachen sind wie die verwendeten Werkzeuge so entworfen, daß sie die Methoden unterstützen.

2.1.2 Software, Programm, Anwendungssoftware

Software ist die Gesamtheit aller Dokumente, die im Laufe einer Produktentwicklung entstehen, sowie deren struktureller Zusammenhang. Die Dokumente müssen permanent gespeichert und verfügbar sein und für die Entwicklung, den Gebrauch und die Wartung des Zielprogramms relevant sein. Zur Software gehören demnach unter anderem das Pflichtenheft, Testdaten, Testprotokolle, Testprogramme, das Benutzungshandbuch und insbesondere das Zielprogramm. Der Begriff *Software-System* wird als Synonym für Software verwendet.

Ein *Programm* ist ein in Maschinensprache übersetzbarer Teil der Software.

Software wird in Anwendungssoftware und Systemsoftware unterteilt. *Anwendungssoftware* ist Software, die entwickelt wird, um spezielle Anwendungsprobleme zu lösen. Zur Anwendungssoftware gehört demnach z.B. ein Lager- oder Buchhaltungsprogramm. *Systemsoftware* dient im Gegensatz zur Anwendungssoftware dazu, um zusammen mit der Hardware die grundlegende Funktionalität des Rechners zur Verfügung zu stellen. Systemsoftware ist z.B. ein Übersetzer oder ein Betriebssystem.

2.1.3 Zielsoftware, Zielprogramm, Zielsprache

Die *Zielsoftware* ist der Teil der Software, der am Ende der Produktentwicklung an den Kunden ausgeliefert wird. Sie enthält als maschinell übersetzbaren Teil das *Zielprogramm* sowie die für das Zielprogramm relevanten Dokumente wie Handbücher und Installationsanweisungen.

Das Zielprogramm ist in einer formalen Sprache, der *Zielsprache*, codiert.

2.1.4 Prototyp, Prototyping, Prototyping-Sprache

Das Wort Prototyp bedeutet im ursprünglichen Sinn „das erste Stück eines Typs oder einer Serie“. Ein Prototyp hat genau diese Bedeutung in vielen konstruktiven Ingenieurwissenschaften wie im Maschinen- oder Fahrzeugbau. Der Bau des Prototyps ist zeit- und kostenintensiv; der Prototyp dient zu Test- und Versuchszwecken und wird anschließend als Vorlage zur eigentlichen Produktion herangezogen. Im Zusammenhang mit der Entwicklung von Software kann diese Interpretation des Begriffes jedoch nicht beibehalten werden. Der Prototyp soll hier möglichst schnell und billig erstellt werden. Ein Prototyp ist also nicht das erste Stück einer Serie, sondern eher eine *Attrappe*, an der z.B. die Handhabung des Programms bereits frühzeitig betrachtet werden kann.

Ein *Prototyp* im Kontext der Software-Entwicklung ist ein Computermodell¹, das einen wesentlichen Aspekt des Zielprogramms zeigt, und ist Teil der Spezifikation.

¹ siehe dazu die Definition in Kapitel 2.2.1

Prototyping ist der Prozeß, in dem ein Prototyp konstruiert und anschließend zyklisch geprüft und modifiziert wird.

Der Prototyp ist in einer formalen Sprache, der *Prototyping-Sprache*, realisiert.

2.2 Die Modellbildung

Die Definition des Begriffs Prototyp sagt aus, daß ein Prototyp ein Modell ist. Diese Tatsache prägt das Wesen des Prototyps und insbesondere auch den Prozeß der Prototypkonstruktion. Der Begriff „Modell“ sowie der Prozeß, in dem ein Modell erstellt wird, werden im folgenden näher betrachtet, um Rückschlüsse für das Prototyping und den Prototyp ziehen zu können.

In der allgemeinen Modelltheorie nach Stachowiak (1973) wird der Modellbegriff untersucht und ein Begriffsnetz für die Modellierung entwickelt. Da der Modellbegriff und der Prozeß des Modellierens grundlegend für das Prototyping sind, werden nachfolgend die Ergebnisse von Stachowiak, soweit sie für diese Arbeit von Bedeutung sind, wiedergegeben.

2.2.1 Modell, Computermodell, Modellieren

Ein *Modell* ist ein Abbild, ein Vorbild oder eine Repräsentation von etwas. Das abgebildete „Etwas“ ist für ein Modell das *Original*. Das Original kann real oder auch nur virtuell sein; es kann insbesondere selbst wieder ein Modell sein. Ein Modell ist ein Abbild oder ein Vorbild des Originals, das in wesentlichen Aspekten, beispielsweise seinem Verhalten oder der äußeren Form, mit dem Original übereinstimmt.

Der Prozeß, dessen Resultat ein Modell ist, wird als *Modellieren*, *Modellbilden* oder auch als *Modellerschaffen* bezeichnet.

Modelle können von unterschiedlicher Art sein. Abbildung 2.2 zeigt die Klassifikation nach Stachowiak. Er unterscheidet zwischen grafischen, psychotechnischen, technischen, biotechnischen und soziotechnischen Modellen¹.

¹ Eine ausführliche Beschreibung der Modellarten ist in Stachowiak (1973) auf den Seiten 159 bis 196 zu finden.

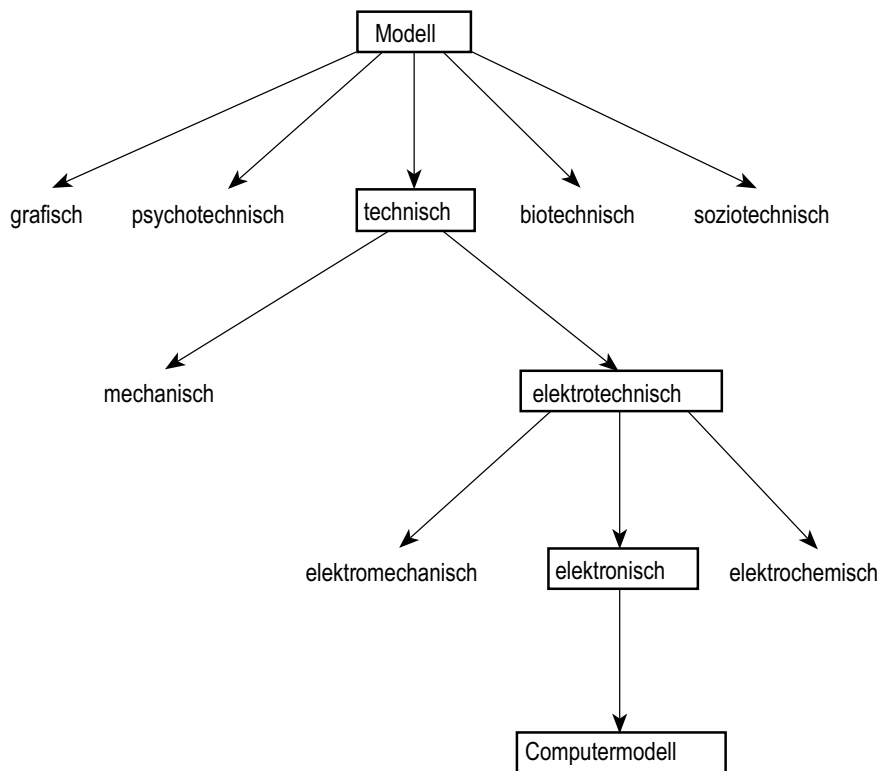


Abb. 2.2: Klassifikation von Modellarten – nach Stachowiak (1973)

Im Zusammenhang mit der Konstruktion von Prototypen sind die technischen Modelle von besonderer Bedeutung. Die technischen Modelle werden in mechanische und elektrotechnische Modelle unterteilt. Zu den mechanischen Modellen gehören die statisch-mechanischen, beispielsweise die maßstäbliche Verkleinerung eines Gebäudes, und die dynamisch-mechanischen Modelle.

Die elektrotechnischen Modelle werden in die elektromechanischen, elektrochemischen und in die elektronischen Modelle unterteilt. Eine spezielle Art der elektronischen Modelle sind die Computermodelle.

Ein *Computermodell* ist ein auf einem Rechner ablauffähiges Programm, dessen Zweck es ist, Informationen über das Original zu erhalten. Solche Programme werden in Luft (1984) als Simulationsprogramme bezeichnet. Beispiele dafür sind Programme zur Wettervorhersage oder Programme zur Flugsimulation. Computermodelle sind immer dann notwendig und

praktikabel, wenn das Original sehr komplex ist, so daß die anschaulichen Verfahren der Modellierung nicht mehr verwendet werden können.

2.2.2 Die Beziehung zwischen Modell und Original

Ein Modell ist ein Abbild von etwas (deskriptives Modell) oder ein Vorbild für etwas (präskriptives Modell). Dieses Etwas – das Original – kann nun selbst wieder ein Modell für ein anderes Original sein. Es gibt vielfältige Gründe, Modelle zu verwenden. Ein Modell wird immer dann benutzt, wenn Untersuchungen am Original nicht möglich oder nicht vertretbar sind. Dies kann der Fall sein, wenn das Original noch nicht existiert oder wenn Untersuchungen am Original zu teuer oder zu gefährlich sind.

Um die Beziehungen zwischen Original und Modell anschaulicher zu machen, werden Original und Modell durch die Menge ihrer Attribute beschrieben¹. Diese Attribute sind die Merkmale und Eigenschaften des Originals oder des Modells. Wählt man als Original z.B. ein Gebäude, so besitzt dieses unter anderem die Attribute Höhe, Fläche und Preis, aber auch die Art und Weise der Konstruktion (z.B. Holz- oder Betonbauweise).

Auf der Basis der Attributbeschreibung von Modell und Original wird der Modellbegriff durch die folgenden drei wesentlichen Merkmale charakterisiert: das *Abbildungsmerkmal*, das *Verkürzungsmerkmal* und das *pragmatische Merkmal*.

Das Abbildungsmerkmal

Da Modelle und Originale durch ihre Attribute beschrieben werden und da ein Modell ein Abbild oder ein Vorbild für ein Original ist, müssen die Attribute des Originals auf die Attribute des Modells abgebildet werden.

Das Verkürzungsmerkmal

Modelle besitzen nicht alle Attribute, die auch das Original besitzt. Für die Modellierung werden nur diejenigen Original-Attribute auf Modell-Attribute abgebildet, die für den Modellerschaffer von Interesse sind. Die Auswahl der abgebildeten Attribute hängt von der Zielsetzung ab, die der

¹ Stachowiak benutzt für diese Menge von Attributen den Begriff "Attributklasse".

Modellbenutzer mit dem Modell verfolgt; die Auswahl ist also immer zweckgebunden.

Die bei der Auswahl der Original-Attribute als nicht relevant klassifizierten Attribute werden *präterierte Attribute* genannt. Die Vereinigung der präterierten Attribute und der bei der Modellierung abgebildeten Attribute des Originals (Attribute des Abbildungsvorbereichs) ergeben zusammen die Menge aller Original-Attribute.

Modelle besitzen neben den abgebildeten Attributen (Attribute des Abbildungsnachbereichs) weitere Attribute, die keine direkte Zuordnung zu Original-Attributen haben. Diese Attribute werden als *abundante Attribute* bezeichnet. Ein Beispiel für ein abundantes Attribut kann z.B. das Material sein, aus dem das Modell hergestellt wird.

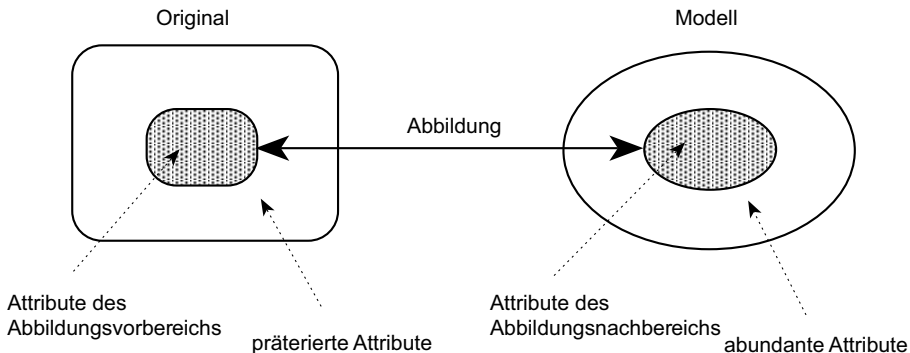


Abb. 2.3: Zusammenhang zwischen Original und Modell

Das pragmatische Merkmal

Ein Modell ist nicht nur Abbild von etwas, sondern es wird, wie auch bereits bei der Auswahl der abzubildenden Attribute angedeutet wurde, immer für einen speziellen Zweck gebildet. Dieser pragmatische Aspekt eines Modells läßt sich in den folgenden Aussagen formulieren:

- Jedes Modell ist ein Abbild von etwas oder ein Vorbild für etwas (Original).
- Jedes Modell wird zu einen speziellem Zweck erstellt.
- Jedes Modell ist ein Abbild für jemanden (Modellbenutzer).
- Jedes Modell wird nur für einen bestimmten Zeitraum benötigt.

Zu diesen Aussagen lassen sich nun vier Fragen formulieren, die gestellt und beantwortet werden müssen, wenn ein Modell erstellt werden soll.

Die vier Grundfragen des Modellierens:

- Was ist das Original des Modells?
- Zu welchem Zweck wird das Modell erstellt, welche Fragestellung soll geklärt werden?
- Wer ist der Modellbenutzer?
- Wann und wie lange soll das Modell benutzt werden?

2.2.3 Simulation

Die Simulation ist ein häufig benutztes Hilfsmittel, um komplexe Systeme zu analysieren. Unter *Simulation* wird in diesem Zusammenhang ein Verfahren verstanden, um Experimente auf einem Rechner an geeigneten Modellen durchzuführen. Das Ziel der Simulation ist, Aussagen über das Verhalten des Systems zu gewinnen. Die Simulation wird also nicht am System selbst, sondern an einem Modell des Systems durchgeführt.

Damit ein Modell für eine Simulation nach der obigen Definition geeignet ist, muß es ein ComputermodeLL sein. Der Vorteil der Simulation an Modellen mit dem Rechner besteht nach Mattern (1989) darin, daß es einfach ist, verschiedene Modellvarianten zu untersuchen und gegeneinander zu bewerten.

2.3 Software-Prototyping

Aufgrund der festgestellten Schwächen bei der Software-Entwicklung nach dem Wasserfallmodell und der damit verbundenen Kritik wurden neue Methoden zur Software-Entwicklung herangezogen. Dabei kann zwischen zwei grundsätzlich verschiedenen Ansätzen unterschieden werden:

Beim *formalen Ansatz* wird versucht, auch die frühen Aktivitäten bei der Software-Entwicklung zu formalisieren, damit unklare oder widersprüchliche Anforderungen mittels formaler Mechanismen entdeckt und beseitigt werden können. Die resultierende formale Spezifikation kann ganz oder teilweise in weiteren formalen Schritten korrektheiterhaltend zum Zielprogramm transformiert werden. Dieser Ansatz berücksichtigt jedoch nicht die in Lehman (1980) formulierte Erfahrung, daß ein Teil der Anforderungen erst dann sichtbar wird, wenn die Software eingesetzt wird.

Der *experimentelle Ansatz* trägt dieser Tatsache Rechnung. Es werden, bevor die Anforderungen fixiert werden, Erfahrungen im Anwendungsgebiet der geplanten Software durch Experimente gewonnen. Prototyping ist eine Form des experimentellen Ansatzes. Beim Prototyping wird dazu sehr früh in der Projektentwicklung ein Computermodell des Zielprogramms (der Prototyp) erstellt, damit die späteren Anwender das Verhalten des Systems bewerten können. Werden dabei Mängel und fehlende Anforderungen entdeckt, so wird der Prototyp verbessert. Dieser Zyklus wird solange durchlaufen, bis alle Personen, die den Prototyp prüfen, mit seinem Verhalten zufrieden sind.

Mithilfe eines Prototyps sollen also die Anforderungen an die geplante Software erkannt werden, die bei Projektbeginn noch nicht oder nicht zutreffend formuliert werden können, weil sie erst durch die Arbeit mit dem Prototyp ersichtlich werden.

Wie Patton (1983) zeigt, wird der Begriff Prototyp in verschiedenen Bedeutungen verwendet. Da er grundlegend für diese Arbeit ist, wird er im folgenden ausführlich erläutert. Die Definition eines Prototyps, wie sie in Abschnitt 2.1.4 angegeben ist, enthält alle Aussagen, die das Wesen eines Prototyps charakterisieren:

- *Ein Prototyp ist ein Computermodell.*
Ein Prototyp ist damit ein ablauffähiges Programm, das ein Modell des Zielprogramms ist. Da ein Prototyp ebenfalls ein Programm ist, kann der Prototyp in die Konstruktion des Zielprogramms einfließen oder sogar zum Zielprogramm werden.
- *Ein Prototyp zeigt immer nur einen Ausschnitt des Zielprogramms.*
Ein Prototyp ist nie das Modell des gesamten Zielprogramms, d.h. es gibt immer präterierte Attribute beim Zielprogramm. Der Prototyp wird konstruiert, um eine bestimmte Fragestellung, die im Zusammenhang mit dem Zielprogramm steht, zu klären.
- *Ein Prototyp gehört zur Spezifikation des Zielprogramms.*
Ein Prototyp wird auf der Basis einer unvollständigen Spezifikation erstellt. Die Spezifikation wird durch einen Prototyp nicht überflüssig. Der Prototyp ergänzt und erhärtet die in der Spezifikation enthaltenen Anforderungen. Da der Prototyp in mehreren Schritten entwickelt und verbessert wird, ist ein Prototyp auch immer Teil der Spezifikation für den nachfolgenden Prototyp.

2.3.1 Klassifizierung von Prototyping

Prototypen werden für unterschiedliche Zwecke und Ziele erstellt. Hallmann (1985), Floyd (1984) und Budde (1987a) geben Kriterien und eine Terminologie an, nach denen die verschiedenen Erscheinungsformen von Prototypen klassifiziert werden können. Prototypen können demnach sinnvoll anhand der folgenden Fragestellungen eingeteilt werden:

- Welches Ziel steht bei der Anwendung von Prototyping im Vordergrund?
- Welcher Aspekt wird durch den Prototyp untersucht?
- Wie detailliert wird der gewählte Aspekt im Prototyp implementiert?

Klassifikation nach dem Kriterium „angestrebtes Ziel“

Floyd (1984) klassifiziert Prototyping nach dem Ziel, das angestrebt wird. Sie unterscheidet drei Arten:

Exploratives Prototyping

Der Prototyp wird mit dem Ziel erstellt, die Systemanalyse zu unterstützen und zu ergänzen. Dies ist immer dann sinnvoll, wenn die Problemstellung nicht klar ist. Der Prototyp soll also helfen, die Problemstellung zu klären. Der Prototyp wird hauptsächlich als Kommunikationsmittel zwischen Entwickler und Auftraggeber verwendet, um die tatsächlichen Systemanforderungen zu ermitteln. Prototypen, die zu diesem Zweck konstruiert werden, werden als *Prototypen im engeren Sinn* oder auch als *eigentliche Prototypen* bezeichnet.

Experimentelles Prototyping

Bei diesem Ansatz dient der Prototyp dazu, mögliche Lösungen für ein bestimmtes Problem zu untersuchen und gegeneinander zu bewerten. Anhand des Prototyps wird untersucht, ob die einzelnen Lösungsvorschläge angemessen sind. Da diese Art besonders bei der Lösung technischer Problemstellungen innerhalb des Entwickler-Teams angewendet werden kann, spricht man bei diesen Prototypen auch von *Labormustern*.

Evolutionäres Prototyping

Der Prototyp wird nicht nur zu Lernzwecken verwendet, sondern bereits als Systemkern konzipiert und im Anwendungsgebiet erprobt. Aus diesem Kern wird in mehreren Iterationsschritten das gesamte Zielprogramm entwickelt. Der erste Prototyp entsteht durch exploratives Prototyping. Beim evolutionären Prototyping existiert keine klare Grenze zwischen Prototyp

und Zielprogramm, da der Prototyp schrittweise zum Zielprogramm wird. Ein auf diese Art und Weise entstehender Prototyp wird auch als *Pilotsystem* bezeichnet.

Ein Pilotsystem ist am weitesten von der eingangs gegebenen Definition eines Prototypen entfernt, da Modell (Prototyp) und Original (Zielprogramm) nicht mehr exakt getrennt werden können.

Neben diesen Zielen, die mit Prototyping erreicht werden sollen, wird in Kieback et al. (1992) darauf hingewiesen, daß Prototypen ebenfalls zum Zweck der Projektakquisition erstellt werden. Diese Prototypen sollen einen ersten Eindruck bei Auftraggeber und künftigen Benutzern erwecken, wie das spätere System aussehen könnte. Diese Prototypen dienen einzig dazu, den Projektauftrag zu erhalten. Sie werden deshalb als *Akquisitions-* oder *Demonstrationsprototypen* bezeichnet.

Wird der Prototyp, nachdem er seine Aufgabe erfüllt hat, weggeworfen, spricht man von *throw-it-away-Prototyping*. Die beim Prototyping gemachten Erfahrungen vervollständigen die Systemspezifikation, der Code wird nicht mehr benötigt.

Klassifikation nach dem Kriterium „modellierter Aspekt“

Prototypen können jeden Aspekt des Zielprogramms modellieren. In der Praxis hat sich aber gezeigt, daß Prototypen häufig in die folgenden Kategorien fallen:

Der Benutzungsschnittstellenprototyp

Immer dann, wenn interaktive Systeme entwickelt werden sollen, kann diese Art von Prototyping vorgesehen werden. Anhand des Prototyps kann der zukünftige Anwender erproben, wie das geplante System benutzt werden soll. Er hat somit die Möglichkeit, Änderungswünsche in den iterativen Prozeß der Prototypentwicklung einfließen zu lassen. Bei dieser Art kann der Anwender besonders gut in die Software-Entwicklung integriert werden.

Der funktionale Prototyp

Dieser Prototyp realisiert einen Teil der Funktionen des geplanten Zielprogramms. Es kann somit überprüft werden, ob die implementierten Funktionen auch den tatsächlichen Anforderungen genügen.

Der Architekturprototyp

Mit einem Architekturprototyp kann die entworfene Architektur des Zielprogramms, die die einzelnen Programmbausteine und deren Beziehungen

definiert, bewertet werden, bevor die einzelnen Bausteine vollständig implementiert sind. Die Bausteine müssen dazu jedoch zumindest rudimentär realisiert sein, damit der Architekturprototyp ausgeführt werden kann.

Klassifikation nach dem Kriterium „Grad der Detaillierung“

Bei der Konstruktion des Prototyps kann der gewählte Schwerpunkt verschieden detailliert implementiert werden. In Budde (1987a) werden zwei unterschiedliche Möglichkeiten genannt:

Horizontales Prototyping

Beim horizontalen Prototyping werden separate Schichten des Zielprogramms erstellt und untersucht. Die Schicht der Benutzungsoberfläche oder die einzelnen Schichten des funktionalen Kerns sind Beispiele dafür. Dies bedeutet, daß ein horizontaler Benutzungsschnittstellenprototyp die gesamte Interaktion zeigt, die darunterliegende Funktionalität wird jedoch nicht berücksichtigt, sondern nur in der Form von Funktionsrümpfen bereitgestellt.

Vertikales Prototyping

Beim vertikalen Prototyping wird ein bestimmter Teil des Zielprogramms für den Prototyp ausgewählt und vollständig implementiert. Ein vertikaler Benutzungsschnittstellenprototyp zeigt nur einen Teil der Interaktion, diesen und insbesondere die dazu benötigte Funktionalität aber vollständig. Diese Vorgehensweise ist bei der Konstruktion von Pilotsystemen üblich.

Die aufgeführten Arten von Prototyping werden in der Praxis nicht in der hier vorgestellten reinen Form vorgefunden. Dies ist dadurch bedingt, daß sich die aufgelisteten Kriterien bei der Konstruktion eines Prototyps nicht so exakt trennen lassen, wie es den Anschein haben könnte. So kann mit einem Benutzungsschnittstellenprototyp nicht sinnvoll gearbeitet werden, wenn die Funktionalität des Systems nicht wenigstens rudimentär implementiert ist. Im Normalfall wird man also Mischformen der hier genannten Arten vorsehen.

2.3.2 Modellierung und Prototyping

Weil der Prototyp ein Modell des Zielprogramms ist, muß beim Prototyping zuerst ein Modellierungsschritt durchgeführt werden. Ein Prototyp besitzt die Merkmale, die auch ein Modell kennzeichnen: ein Prototyp ist ein Abbild des Zielprogramms (Abbildungsmerkmal), er zeigt nicht alle Aspekte des Zielprogramms (Verkürzungsmerkmal), er wird für einen

bestimmten Zweck erstellt (pragmatisches Merkmal). Daraus folgt, daß die vier Grundfragen des Modellierens auch berücksichtigt werden müssen, wenn ein Prototyp erstellt werden soll. Nachfolgend werden diese im Kontext von Prototyping beantwortet.

Was ist das Original des Prototyps?

Das Original des Prototyps, das geplante Zielprogramm, ist zum Zeitpunkt seiner Konstruktion nur virtuell vorhanden. Es existiert in den Köpfen des Kunden und der Entwickler. Das Zielprogramm soll erst aufgrund der Erfahrungen mit dem Prototyp erstellt werden. Die Problem- und Anforderungsbeschreibung für das Zielprogramm ist somit die Grundlage, um einen Prototyp zu erstellen. Daneben existieren sowohl beim Kunden als auch beim Entwickler Ideen und Vorstellungen, die nicht schriftlich formuliert sind, aber bei der Konstruktion des Prototyps einfließen.

Welche Fragestellung soll der Prototyp klären?

Ein Prototyp soll dazu beitragen, eine bestimmte Fragestellung, die für die Entwicklung des Zielprogramms relevant ist, zu beantworten. Diese Fragestellung muß nach einer Analyse der Anforderungsbeschreibung definiert werden. Der Prototyp wird ausschließlich gegen diese Fragestellung geprüft. Aus der Klassifikation von Prototypen ergeben sich vier wesentliche Ziele, die mit Prototypen erreicht werden sollen:

- Es sollen gestellte Anforderungen validiert und nicht erkannte Anforderungen gefunden werden (der eigentliche Prototyp).
- Es sollen Lösungsalternativen für eine Problemstellung untersucht werden oder es soll die technische Realisierbarkeit einer Lösung gezeigt werden (das Labormuster).
- Es soll gezeigt werden, daß die realisierte Lösung praktikabel ist und den Anforderungen im Einsatz genügt (das Pilotsystem).
- Der Prototyp soll bereits vor Projektbeginn einen möglichen Lösungsansatz für die Problemstellung zeigen, um den Auftrag für das Projekt zu erhalten oder die weitere Finanzierung des Projekts zu sichern.

Welchen Aspekt soll der Prototyp modellieren?

Der modellierte Aspekt ist von der Fragestellung abhängig, die mit dem Prototyp geklärt werden soll. Sie bestimmt den Ausschnitt des Zielprogramms, der im Prototyp abgebildet wird. Dieser Ausschnitt ist der

Abbildungsvorbereich der Modellierung. Daneben ist es ebenfalls notwendig, die Aspekte zu formulieren, die explizit nicht im Prototyp modelliert werden. Diese Aspekte sind in der Terminologie der Modelltheorie die präterierten Attribute des Zielprogramms. Genauso wichtig ist es, die Merkmale des Prototyps anzugeben, die nicht auf das Zielprogramm übertragbar sind – die abundanten Attribute des Prototyps. Bei einem Benutzungsschnittstellenprototyp kann z.B. ein abundantes Attribut die Sprache sein, in der der Prototyp realisiert ist.

Wer sind die Benutzer des Prototyps?

Die Benutzer des Prototyps sind die Personen, die den Prototyp erproben und prüfen. Welche Personengruppen dies sind, ist von der untersuchten Fragestellung abhängig. Beim eigentlichen Prototyp und beim Pilotsystem sind dies in der Regel die Anwender. Bei Labormustern sind es die Entwickler oder die DV-Spezialisten des Auftraggebers.

Wie hoch ist der Aufwand für den Prototyp?

Für jede Prototypentwicklung muß der zur Verfügung gestellte Zeitraum und das bereitgestellte Budget angegeben werden. Ein Prototyp muß dabei erheblich schneller und billiger erstellt werden als der entsprechende Ausschnitt des Zielprogramms.

Die Beantwortung dieser Fragen ergibt die *Spezifikation des Prototyps*. Sie ist die Grundlage, auf der ein Prototyp anschließend entworfen, implementiert und bewertet wird.

2.3.3 Der allgemeine Entwicklungszyklus beim Prototyping

Software-Entwicklung ist ein technischer Prozeß, der unterschiedlich gestaltet werden kann. Das Software-Engineering versucht diesen Prozeß zu beschreiben, indem unterschiedliche Prozeßmodelle definiert werden. In Ludewig (1989) ist eine kurze Charakterisierung der zur Zeit in der Praxis und in der Literatur bekannten Prozeßmodelle enthalten.

Es gibt verschiedene Ansätze, Prototyping in den traditionellen Software-Entwicklungsprozeß, wie er durch das Wasserfallmodell charakterisiert wird, zu integrieren. Wie Kieback et al. (1992) feststellen, führt dies jedoch zu Schwierigkeiten, da die traditionelle Vertragsgestaltung entlang definierter Meilensteine nicht besonders geeignet ist, um Prototypen bei der

Entwicklung miteinzuplanen. Auf der anderen Seite gibt es Ansätze, neue prototypspezifische Prozeßmodelle zu entwickeln. Beispiele für diesen Ansatz sind in Budde et al. (1992c) und in Bischofberger (1989) zu finden. Jede Prototypentwicklung kann jedoch durch den im folgenden erläuterten Arbeitszyklus modelliert werden.

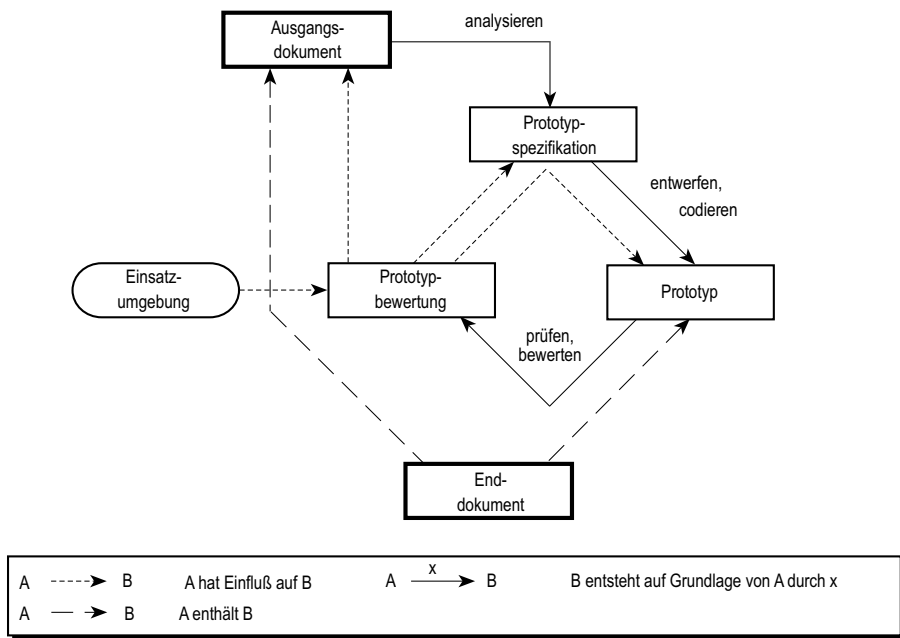


Abb. 2.4: Der allgemeine Entwicklungszyklus beim Prototyping

Die Ausgangssituation beim Prototyping ist dadurch gekennzeichnet, daß Informationen, die für die Entwicklung der geplanten Software notwendig sind, fehlen oder nur vage und unvollständig beschrieben werden können. Diese Situation ist in der Regel in einem Dokument fixiert, dem *Ausgangsdokument*. Dieses Dokument kann z.B. eine informale Problembeschreibung oder eine Modulspezifikation sein. Auf der Grundlage dieses Ausgangsdokuments wird die Prototypspezifikation erstellt. Sie enthält Angaben zum Zweck des Prototyps, zu den an der Entwicklung beteiligten Personengruppen, zur Entwicklungsdauer und zu den Kosten. Aufgrund der Prototypspezifikation wird der Prototyp entworfen und implementiert. Anschließend beginnt der Zyklus, bei dem der Prototyp benutzt, bewertet und modifiziert wird. Geprüft wird der Prototyp gegen die untersuchte

Fragestellung. Es sind daran die Personengruppen beteiligt, die in der Prototypspezifikation dafür vorgesehen wurden.

Dieser Zyklus wird solange durchlaufen, bis der untersuchte Aspekt im Prototyp entweder so modelliert ist, daß alle Personen, die an der Prüfung beteiligt sind, damit zufrieden sind, oder bis der vorgesehene Zeit- und Kostenrahmen ausgeschöpft ist. Im letzten Fall muß der aktuelle Zustand entweder akzeptiert oder weiterer Aufwand für die Prototypentwicklung budgetiert werden.

Eine entscheidende Rolle spielt die Bewertung des Prototyps. Das Verhalten des Prototyps wird dabei in der Einsatzumgebung erprobt. Die Bewertung kann dazu führen, daß Änderungen am Anfangsdokument, an der Prototypspezifikation und natürlich auch am Prototyp direkt notwendig werden. Am Ende der Prototypentwicklung erhält man das *Enddokument*, das aus dem veränderten Anfangsdokument und aus dem Prototyp „besteht“. Der Prototyp, der zuerst ein Abbild des Originals war, ist, wie in Ludewig (1989) dargestellt wird, durch den Zyklus von Prüfung und Modifikation zu einem Vorbild für das Original geworden.

Kapitel 3

Software-Modellierung und Software-Architekturen

In diesem Kapitel wird die Modellierung der Software diskutiert. Es wird gezeigt, welche Modellierungsebenen existieren und wie diese miteinander in Verbindung stehen.

Anschließend wird ein Modellierungsansatz für Software vorgestellt, der auf den Dokumenten basiert, die während der Software-Entwicklung erstellt werden. Dieser Modellierungsansatz führt zu einem allgemeinen Architekturbegriff, der auf die Software übertragen wird. Abschließend wird eine Klassifikation der Software-Modellierungsarten angegeben, die zum Software-Architektur-Prototyping führt.

3.1 Prozeßmodelle und Software-Modelle

Software ist nach der in Kapitel 2 gegebenen Definition die Menge aller Dokumente und deren Strukturierung, die im Laufe einer Software-Entwicklung entstehen. Die Dokumente müssen permanent gespeichert und verfügbar sowie für die Entwicklung, den Einsatz und die Wartung des Zielprogramms relevant sein.

Software-Entwicklung ist dementsprechend der Prozeß, in dem die Software entsteht.

Beide Aspekte, sowohl der Software-Entwicklungsprozeß als auch dessen Ergebnis, können modelliert werden. Dies führt auf der einen Seite zu Modellen der Software-Entwicklung, die *Prozeßmodelle* genannt werden und in Ludewig (1989) zusammenfassend beschrieben sind, und auf der anderen Seite zu Modellen der Software selbst. Sowohl Prozeßmodelle als auch Software-Modelle können Abbilder oder Vorbilder für die entsprechenden Originale sein. Beide Modellarten zeigen jeweils nur die für die Modellierung relevanten Attribute des Originals.

3.2 Ebenen der Modellierung

Bei der Modellierung der Software lassen sich drei verschiedene Ebenen identifizieren: die *Schemaebene*, die *Modellebene* und die *Ausprägungsebene*. Abbildung 3.1 zeigt die Beziehungen zwischen den Modellierungsebenen.

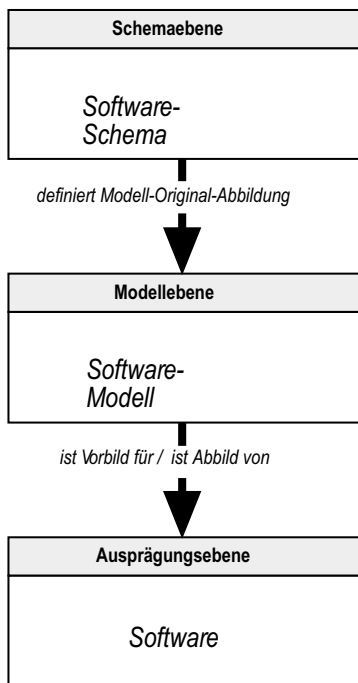


Abb. 3.1: Die Modellierungsebenen der Software

3.2.1 Die Schemaebene

In der *Schemaebene* werden alle *Baustein-* und *Beziehungsarten* festgelegt, die zur Verfügung stehen, um Software-Systeme zu modellieren. Dies wird als *Software-Schema* bezeichnet. Ein Software-Schema definiert weiterhin Konsistenzbedingungen, die für alle nach dem Schema erstellten Modelle gelten.

Ein Software-Schema definiert somit, welche Möglichkeiten für die Software-Modellierung vorgesehen sind. Mit einem Software-Schema werden neben der Abbildung von Original- auf Modellattribute auch implizit die präterierten Original-Attribute definiert.

Der Teil des Software-Schemas, der vorgesehen ist, um das Zielprogramm zu modellieren, wird als *Zielprogramm-* oder auch nur als *Programmschema* bezeichnet.

Ein Software-Schema wird in einer dafür geeigneten Sprache, der *Schema-beschreibungssprache*, formuliert. Da ein Software-Schema Baustein- und Beziehungsarten definiert, kann eine Entity-Relationship-Notation¹ verwendet werden.

3.2.2 Die Modellebene

In der *Modellebene* wird das Modell einer Software mit den Mitteln des zugrundeliegenden Software-Schemas erstellt.

Ein Software-Modell wird in einer Sprache notiert, mit der Bausteine und Beziehungen formuliert werden können. Diese Sprache ist an das Software-Schema angepaßt und wird *Software-Modellierungssprache* genannt. Es ist möglich und mitunter sinnvoll, isomorphe Software-Modellierungssprachen, grafisch oder textuell, für ein Software-Schema zu entwickeln und bei der Modellierung zu verwenden.

3.2.3 Die Ausprägungsebene

Die konkrete Software entsteht in der *Ausprägungsebene*, wobei das erstellte Modell das Vorbild ist. Damit das Software-Modell als Vorbild verwendet werden kann, muß eine Abbildung zwischen den im Software-Schema definierten Baustein- und Beziehungsarten und Elementen der Ausprägungsebene angegeben werden. Die Baustein- und Beziehungsarten des Programmschemas müssen dazu auf geeignete Konstrukte der Zielsprache oder des Betriebssystems abgebildet werden.

Die Software zeigt jedoch zusätzliche Merkmale, die für das Modell nicht relevant sind. Dies sind, in der Terminologie der allgemeinen Modelltheorie,

¹ Der Begriff „Entity-Relationship“ wird im weiteren durch das übliche Akronym „ER“ abgekürzt.

die präterierten Merkmale des Originals. Da diese Merkmale für das Modell nicht wichtig sind, können sie auch nicht in der Software-Modellierungssprache formuliert werden.

3.2.4 Ein einfaches Beispiel

Abbildung 3.2 zeigt an einem einfachen Programmschema, wie die beschriebenen Ebenen zueinander in Beziehung stehen.

Das Programmschema stellt die Bausteinarten „Modul“ und „Hauptprogramm“ sowie die Beziehungsarten „benutzt (Modul, Modul)“ und „benutzt (Hauptprogramm, Modul)“ bereit. Für die benutzt-Beziehungsart zwischen Modulen gilt die Konsistenzbedingung, daß die Beziehungsart irreflexiv ist, d.h. ein Modul darf sich selbst nicht benutzen.

Entsprechend diesem Schema kann ein Programmmodell, bestehend aus den Modulen „A“ und „B“, dem Hauptprogramm „C“ sowie den Beziehungen „C benutzt A“, „C benutzt B“ und „B benutzt A“ erstellt werden.

In der Ausprägungsebene werden zwei Programme gezeigt: das erste ist in der Programmiersprache Ada, das zweite in der Programmiersprache C notiert. Die Programme sind nach dem Vorbild des Modells erstellt, wobei die Baustein- und Beziehungsarten des Programmschemas folgendermaßen auf Elemente der Ausprägungsebene abgebildet sind.

Schemakonstrukt	Ada	C
Modul Hauptprogramm benutzt (Modul, Modul) benutzt (Hauptprogramm, Modul)	package procedure with-clause with-clause	Dateien: <Modulname>.h / <Modulname>.c main-function #include <Modulname>.h #include <Modulname>.h

Die Module des Programms definieren zusätzlich zum Modell auch die Operationen, die sie exportieren. Diese Information ist für das Programmmodell nicht relevant, so daß im Programmschema keine Bausteinart „Operation“ und keine Beziehungsart „exportiert“ zwischen Modul und Operation vorgesehen wurde.

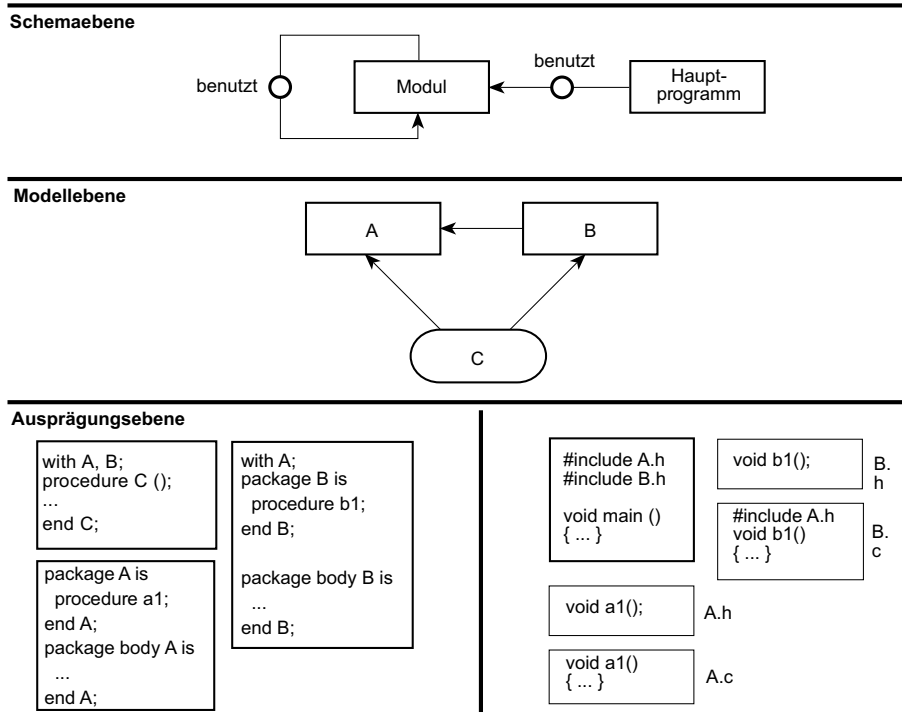


Abb. 3.2: Ein Beispiel für die Modellierungsebenen

3.3 Die dokumentenbasierte Modellierung der Software

Im folgenden wird ein Ansatz für die Software-Modellierung vorgestellt. Dieser geht davon aus, daß während der Software-Entwicklung stetig Dokumente produziert werden. Der Ansatz ist unabhängig von einem bestimmten Prozeßmodell.

3.3.1 Software-Entwicklung ist Dokumentenentwicklung

Es gibt verschiedene Arten von industriellen Software-Projekten. Frühauf (1988) unterscheidet Auftragsprojekte, interne Projekte und Entwicklungsprojekte. Es gibt jedoch immer ein Dokument, mit dem Projekte – unabhängig von welcher Art sie sind – initiiert werden. Dieses Dokument ist beispielsweise bei einem Auftragsprojekt der Vertrag, der zwischen Kunde

und Entwickler abgeschlossen wird. Das Dokument, das ein Software-Projekt initiiert, kann als das *Basisdokument* der Software betrachtet werden.

Alle folgenden Dokumente entstehen letztlich aufgrund dieses ersten Dokuments, *unabhängig* davon, nach welcher konkreten Vorgehensweise entwickelt wird. Jedes Dokument ist das Ergebnis einer Tätigkeit während der Software-Entwicklung und wird für einen speziellen Verwendungszweck erstellt. So werden Dokumente erstellt, um Anforderungen zu beschreiben, um Testfälle aufzunehmen oder um als Programmbaustein verwendet zu werden.

Ein Software-Schema, das auf dieser Betrachtungsweise der Software-Entwicklung basiert, muß Möglichkeiten zur Verfügung stellen,

- damit die entstehenden Dokumente modelliert und
- Beziehungen zwischen den Dokumenten eingerichtet werden können.

3.3.2 Grundlegende Beziehungen zwischen Dokumenten

Alle Dokumente, die im Laufe einer Entwicklung entstehen, können nach dem Zeitpunkt geordnet werden, zu dem sie entstanden sind. Dabei bilden die Basisdokumente den Ausgangspunkt für die Software-Entwicklung¹.

Die zeitliche Reihenfolge, in der die Dokumente entstehen, kann mit der allgemeinen Beziehungsart *ist_entstanden_nach* zwischen Dokumenten modelliert werden. Die Beziehung „D2 ist_entstanden_nach D1“ drückt beispielsweise aus, daß das Dokument mit dem Namen „D2“ später als das Dokument „D1“ erzeugt wurde.

Neben der zeitlichen Abfolge können die Dokumente auch danach strukturiert werden, welchen Einfluß sie auf das Entstehen anderer Dokumente haben. Diese Beziehungsart wird *ist_entstanden_aufgrund* genannt. Sie spezialisiert die Beziehungsart „ist_entstanden_nach“, da sie zusätzlich zur temporalen auch immer eine inhaltliche Abhängigkeit zwischen den Dokumenten ausdrückt. Die Beziehung „D2 ist_entstanden_aufgrund D1“ modelliert entsprechend folgendes:

- „D2“ ist später entstanden als „D1“.

¹ Zu den Basisdokumenten zählen neben dem Projektauftrag z.B. auch Richtlinien, die für alle Projekte gelten.

- „D2“ wurde erstellt, weil entweder bei der Arbeit mit „D1“ festgestellt wurde, daß „D2“ benötigt wird, oder weil „D1“ vorschreibt, daß „D2“ erstellt werden muß.

Die bei der Software-Entwicklung entstehenden Dokumente und Beziehungen bilden abstrakt betrachtet einen zusammenhängenden Graph, dessen Knoten die Dokumente und dessen Kanten die Beziehungen zwischen den Dokumenten sind.

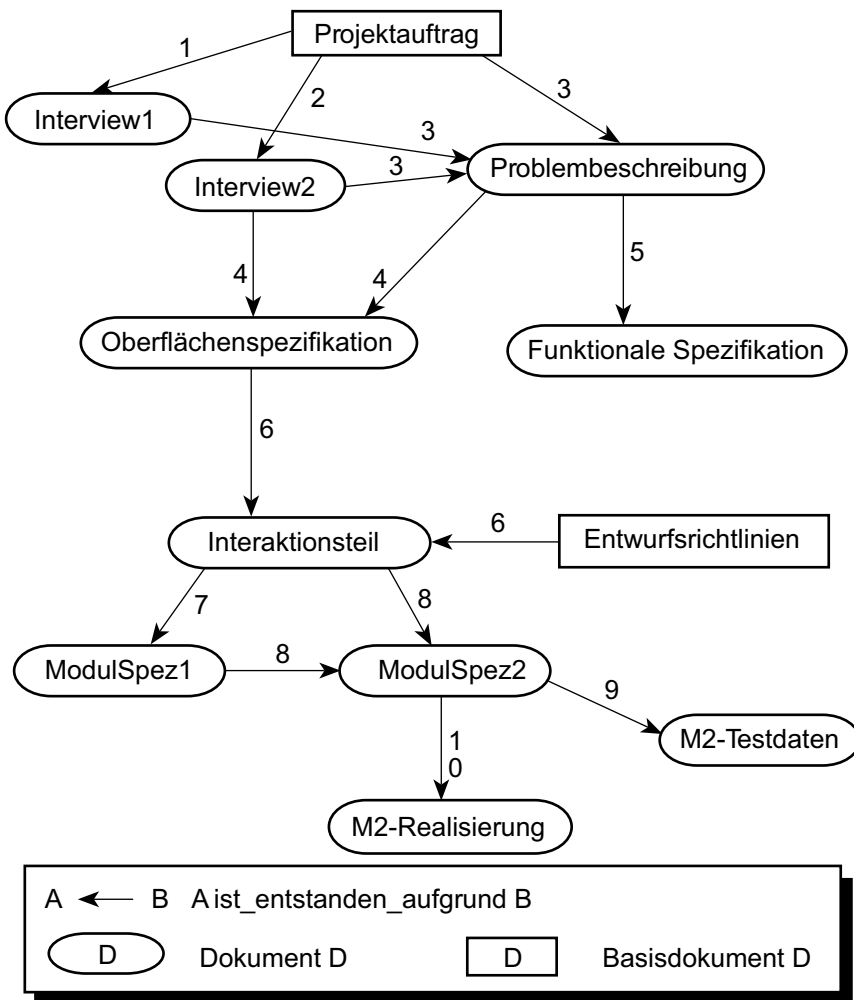


Abb. 3.3: Beispiel für die Entstehungsreihenfolge von Dokumenten

Abbildung 3.3 zeigt beispielhaft einen Ausschnitt aus der Entstehungsreihenfolge von Dokumenten. Die Numerierung der abgebildeten Beziehungen entspricht der Entstehungsreihenfolge.

Die Abbildung kann folgendermaßen interpretiert werden:

- 1, 2 Nachdem der Projektauftrag erteilt wurde, werden zwei Interviews beim Kunden durchgeführt. „Interview1“ mit einem Fachingenieur und „Interview2“ mit einem Anwender.
- 3 Aufgrund der Informationen, die durch die Interviews gewonnen wurden, und den Angaben im Projektauftrag wird die Problembeschreibung erstellt.
- 4, 5 Die Problembeschreibung und die Aussagen des Anwenders in „Interview2“, die besonders die Programmbedienung betreffen, führen zur Spezifikation der Benutzungsschnittstelle. Die funktionale Spezifikation wird aufgrund der Problembeschreibung erstellt.
- 6 Aufgrund der Entwurfsrichtlinien, die für interaktive Anwendungen gelten, und der Oberflächenspezifikation, wird ein Entwurfsdokument für den Interaktionsteil der Anwendung erstellt.
- 7 Dieses ist die Grundlage für die Spezifikation eines Moduls, das die Interaktion realisieren soll (ModulSpez1).
- 8 Ein weiteres Modul, das als Hilfsmodul einen Teil der Interaktion abwickeln soll, wird spezifiziert (ModulSpez2).
- 9, 10 Testdaten werden für „Modul 2“ erstellt und das Modul selbst realisiert.

3.4 Die Entwicklungsgeschichte der Dokumente

Die bei der Software-Entwicklung entstehenden Dokumente entwickeln sich während des Prozesses weiter. Im folgenden wird diskutiert, wie die Entwicklungsgeschichte der Dokumente in den bisher vorgestellten Ansatz zur Software-Modellierung integriert werden kann. Die grundlegenden Ideen dazu basieren auf dem in Tichy (1988) vorgestellten Ansatz, um die Revisionen- und Variantenbildung von Dokumenten zu beschreiben.

3.4.1 Einführung

Dokumente entwickeln sich im Prozeß der Software-Entwicklung weiter, weil sie am Anfang unvollständig sind, Fehler enthalten oder an neue Er-

kenntnisse und Anforderungen angepaßt werden müssen. Die Weiterentwicklung der Dokumente kann auch als Wartung bezeichnet werden. Lientz (1978) identifiziert drei Situationen, die es erforderlich machen, daß ein Dokument weiterentwickelt werden muß:

- Das Dokument enthält Fehler, die beseitigt werden müssen (corrective maintenance).
- Das Dokument muß an neue Anforderungen angepaßt werden (adaptive maintenance).
- Das Dokument wird inhaltlich verbessert (perfective maintenance).

Diese Situationen existieren für alle Arten von Dokumenten, sowohl für Code-, Text- als auch für Grafikdokumente. Die Entwicklungsgeschichte der Dokumente ist ein Thema, das in zahlreichen Arbeiten aus dem Gebiet der Software- Konfigurationsverwaltung behandelt wird. Verschiedene Ansätze dazu sind in Winkler (1988) enthalten.

Für alle Ansätze gilt, daß immer, wenn ein Dokument verändert wird, ein neues Dokument entsteht. Dabei wird unter einer Veränderung nicht jede einzelne Modifikation an einem Dokument verstanden, sondern die Gesamtheit aller Modifikationen, die notwendig sind, um ein Dokument im Sinn der vorhin genannten Wartungsaktivitäten weiterzuentwickeln.

Im weiteren wird das Dokument, das verändert wird, als *Ausgangsdokument* bezeichnet. Das Dokument, bei dem die Entwicklungsgeschichte beginnt, wird *initiales Dokument* genannt.

Im Zusammenhang mit der Entwicklungsgeschichte der Dokumente werden in der Literatur die Begriffe *Version*, *Revision*, *Variante* und *Konfiguration* verwendet. Tichy (1988) bezeichnet alle Dokumente, die aus einem initialen Dokument durch Veränderungen entstehen, als Versionen oder Versionsgruppe. Revisionen und Varianten sind spezielle Arten von Versionen. Nachfolgend werden die Revisionen- und Variantenbildung im einzelnen betrachtet.

3.4.2 Die Revisionenbildung

Eine Revision ist ein Dokument, das aus dem Ausgangsdokument hervorgeht und statt dessen verwendet werden soll. Die Beziehungsart zwischen den beiden Dokumenten, dem Ausgangsdokument und der Revision, wird *ist_Revision_von* genannt. Sie ist eine Spezialisierung der allgemeinen Beziehungsart „ist_entstanden_aufgrund“, da sie ausdrückt,

daß die Revision aus dem Ausgangsdokument entstanden ist, nachdem dieses im oben beschriebenen Sinn verändert wurde. Zu einem Dokument gibt es maximal eine Revision, die direkt aus ihm hervorgegangen ist. Auf der anderen Seite bezieht sich eine Revision immer auf genau ein Ausgangsdokument.

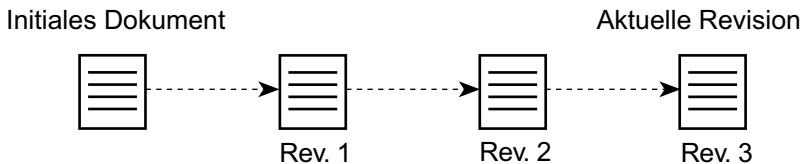


Abb. 3.4: Revisionenbildung

Abbildung 3.4 zeigt schematisch die Revisionenbildung. Durch die Beziehungsart „ist_Revision_von“ wird ein Baum aufgespannt, der zwei Knoten vom Grad eins hat (Wurzel und Blatt); alle anderen Knoten haben den Grad zwei. Die Wurzel des Baumes ist das initiale Dokument. Die restlichen Knoten sind Revisionen. Das Blatt des Baumes ist die *aktuelle Revision*.

3.4.3 Die Variantenbildung

Eine *Variante* ist ein Dokument, das als Alternative zum Ausgangsdokument verwendet werden kann, obwohl sich die beiden Dokumente in Details unterscheiden. Werden diese Details nicht betrachtet, können die Dokumente nicht unterschieden werden. Die unterschiedlichen Details definieren den varianten Teil. Die abstrakte Sicht auf die Dokumente, bei der die Details vernachlässigt werden, definiert den invarianten Teil.

Varianten zu Code-Dokumenten werden beispielsweise dann erstellt, wenn unterschiedliche Betriebssysteme oder Fenstersysteme berücksichtigt werden sollen. Varianten zu Textdokumenten werden dann erstellt, wenn diese z.B. in unterschiedlichen Sprachen verfügbar sein sollen.

Die Beziehungsart zwischen den beteiligten Dokumenten, dem Ausgangsdokument und der Variante, wird *ist_Variante_zu* genannt. Nachfolgend ist eine Entwicklungsgeschichte abgebildet, bei der Revisionen und Varianten erzeugt wurden. Diese beginnt beim initialen Dokument.

Zu einem Dokument kann es beliebig viele Varianten geben, eine Variante ist jedoch immer aus genau einem Ausgangsdokument entstanden. Die Beziehungsart ist ebenfalls eine Spezialisierung der Beziehungsart

„ist_entstanden_aufgrund“. Da Varianten ebenfalls Dokumente sind, können sie selbst wieder in Revisionen vorliegen.

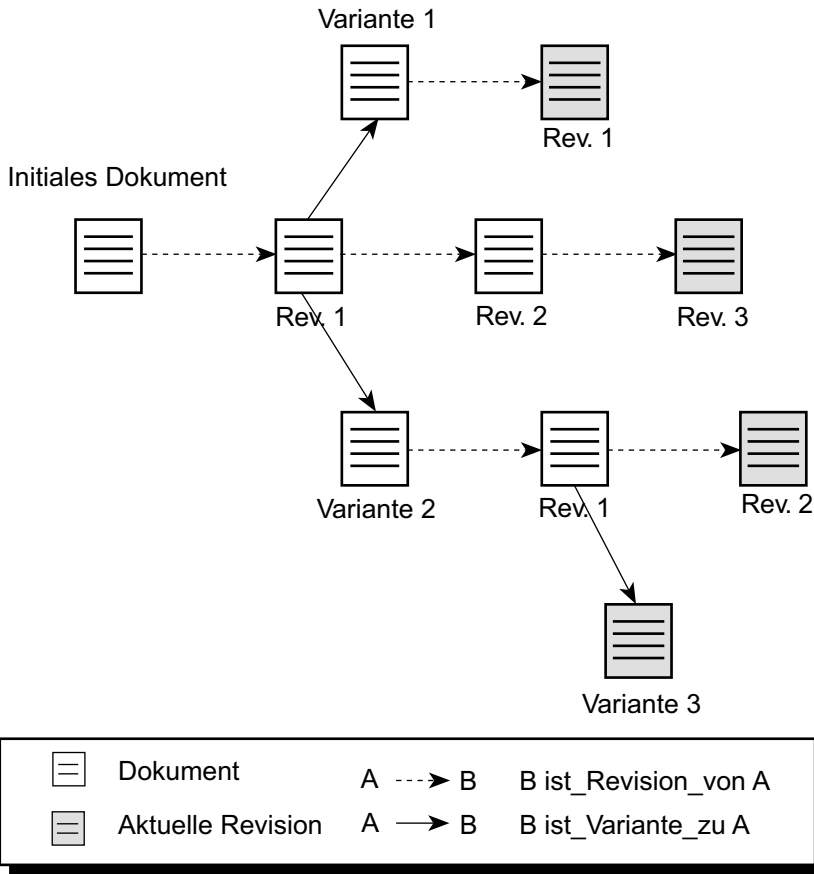


Abb. 3.5: Revisionen- und Variantenbildung

Die Struktur, die durch die beiden Beziehungsarten „ist_Revision_von“ und „ist_Variante_zu“ gebildet wird, ist ein Baum, dessen Wurzel das initiale Dokument ist. Die Anzahl der verfügbaren Varianten ist um eins höher als die Anzahl der ist_Variante_zu-Kanten des Baumes. Der Baum selbst ist ein Teilgraph des Graphen, der durch alle Dokumente und deren Beziehungen aufgespannt wird.

3.4.4 Die Versionenbildung im Detail

Betrachtet man den Prozeß, wie eine Version, d.h. entweder Revision oder Variante, erstellt wird genauer, so ergibt sich der in der folgenden Abbildung gezeigte Ablauf.

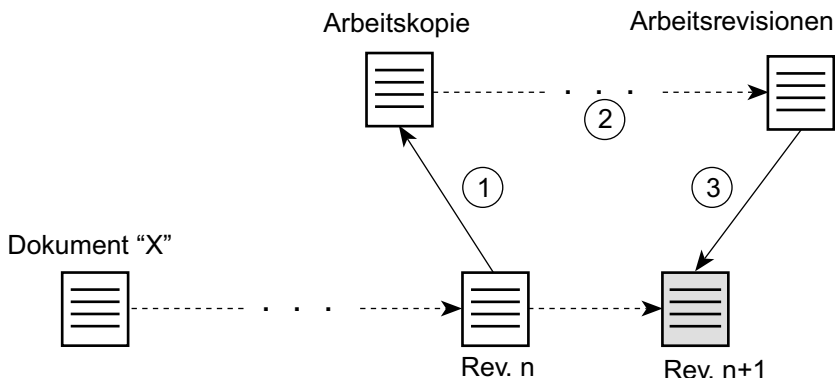


Abb. 3.6: Die Versionenbildung im Detail

Ein Dokument „X“ liegt in der Revision „n“ vor. Es ist für die Benutzer zugänglich und wird verwendet, wobei es jedoch nicht verändert werden kann. Aufgrund neuer Erkenntnisse muß das Dokument „X“ weiterentwickelt werden.

Dazu wird im ersten Schritt eine Arbeitskopie erzeugt, in der die erforderlichen Änderungen vorgenommen werden sollen. Das Original (Dokument „X“ in Revision „n“) bleibt weiterhin zugänglich. Im zweiten Schritt werden die Modifikationen in der Arbeitskopie durchgeführt. Diese erfolgen in der Regel nicht in einem Arbeitsschritt, sondern in mehreren getrennten Schritten. Dabei entstehen Arbeitsrevisionen. Die letzte Arbeitsrevision wird im dritten Schritt, nachdem eventuelle Prüfungen vorgenommen worden sind, die aktuelle Revision „n+1“ in der Entwicklungsschicht des Dokuments „X“. Sie wird für die Benutzer zugänglich gemacht. Die Revision „n“ kann ebenfalls zugänglich bleiben, wenn es z.B. Benutzer gibt, die zwar Revision „n“, nicht aber Revision „n+1“ verwenden können, oder sie wird unzugänglich gemacht.

3.5 Ein dokumentenbasiertes Software-Schema

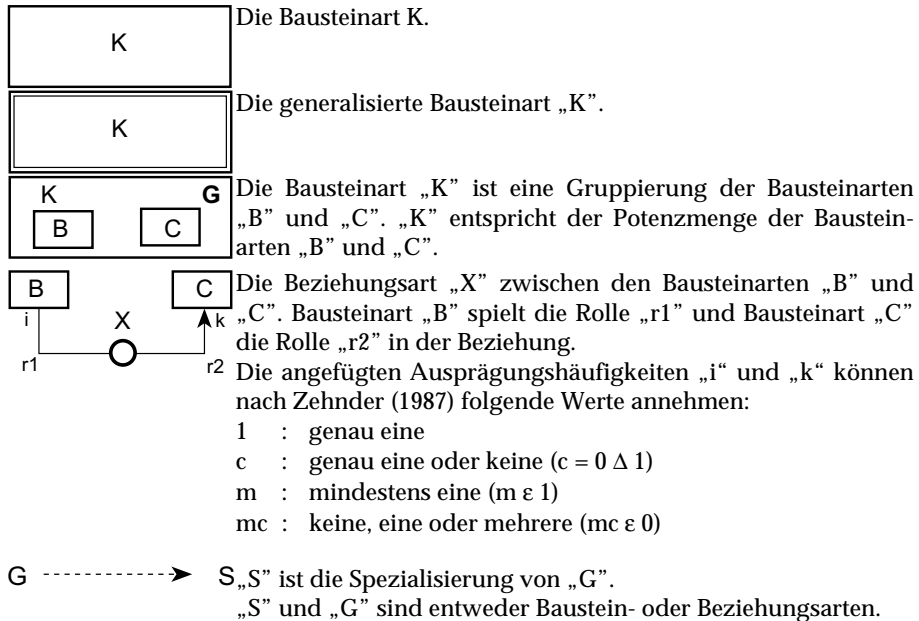
In diesem Abschnitt wird ein Software-Schema definiert, das die Überlegungen zur dokumentenbasierten Software-Modellierung umsetzt. Dieses Schema wird im weiteren als *DS-Schema* (**D**okumentenbasiertes **S**oftware-Schema) bezeichnet.

Bevor das DS-Schema vorgestellt wird, wird eine Notation eingeführt, mit der das DS-Schema und alle anderen Software-Schemata definiert werden, die in dieser Arbeit vorkommen.

3.5.1 Eine Notation für Software-Schemata

Ein Software-Schema kann bis auf die Konsistenzbedingungen übersichtlich in einer ER-Notation dargestellt werden. Die im ER-Ansatz definierten Modellierungsmöglichkeiten, wie Gruppierung oder Spezialisierung, werden in Matheis (1990) zusammenfassend erläutert und in der dort beschriebenen Bedeutung verwendet.

Mit der folgenden grafischen Notation werden Software-Schemata definiert:



Im Text werden folgende Darstellungskonventionen verwendet:

- Bausteinararten werden durch ein Substantiv im Singular bezeichnet und in Schrägschrift gesetzt.
- Beziehungsarten werden durch ein Verb bezeichnet und in der Form *Beziehungsname (Bausteinar₁, Bausteinar₂)* notiert sowie in Schrägschrift gesetzt.
- Konkrete Beziehungen der Beziehungsart *Beziehungsname (Bausteinar₁, Bausteinar₂)* werden in der Infix-Notation *X_Beziehungsname_Y* dargestellt, wobei „X“ eine Ausprägung der Bausteinar₁ und „Y“ eine Ausprägung der Bausteinar₂ ist. Konkrete Beziehungen werden ebenfalls in Schrägschrift dargestellt.

Konsistenzbedingungen für Software-Schemata

Konsistenzbedingungen können nach Zehnder (1987) in zwei Gruppen eingeteilt werden:

- Konsistenzbedingungen, die mit den Möglichkeiten des verwendeten Modells formuliert werden können.
- Konsistenzbedingungen, die zusätzlich angegeben werden müssen.

Diese Einteilung gilt auch für Konsistenzbedingungen, die für Software-Schemata formuliert werden.

Zur ersten Gruppe der Konsistenzbedingungen zählen nach Matheis (1990):

- Ausschlußbedingungen für Bausteinararten, die in mehreren Beziehungsarten auftreten (z.B. ein Programmbaustein kann entweder einen anderen Programmbaustein benutzen oder von diesem erben).
- Minimale und maximale Kardinalitäten von Beziehungsarten (z.B. ein Dokument kann kein Dokument oder bis maximal zehn andere Dokumente referenzieren)
- Spezielle Eigenschaften der Beziehungsarten wie reflexiv, irreflexiv, transitiv, symmetrisch, antisymmetrisch oder azyklisch.

Die Konsistenzbedingungen für ein Software-Schema werden bis auf die Kardinalitäten nicht im Diagramm dargestellt, sondern zusätzlich textuell formuliert.

3.5.2 Das DS-Schema

Abbildung 3.7 zeigt das DS-Schema, dargestellt in der grafischen Notation.

Die Baustein- und Beziehungsarten des DS-Schemas

Das DS-Schema stellt zur Software-Modellierung ausschließlich die Bausteinart *Dokument* zur Verfügung. Damit werden alle Dokumente der Software modelliert. Jedes Dokument kann mithilfe des Attributs „Typ“ näher charakterisiert werden. Dadurch ist es möglich, unterschiedliche Arten von Dokumenten, wie Anforderungs- oder Code-Dokumente, zu modellieren.

Das DS-Schema kennt folgende Beziehungsarten:

Die Beziehungsart *steht_in_Beziehung_zu (Dokument, Dokument)*

Sie ist die generelle Beziehungsart des Schemas. Mit ihr können beliebige Beziehungen zwischen Dokumenten modelliert werden. Damit eine Beziehung näher charakterisiert werden kann, besitzt die Beziehungsart das Attribut „Typ“. Somit kann beispielsweise modelliert werden, daß ein Dokument ein anderes Dokument referenziert oder Testfälle für ein Dokument enthält.

Konsistenzbedingungen:

- Ein Dokument kann Beziehungen zu beliebig vielen anderen Dokumenten haben und kann von beliebig vielen Dokumenten referenziert werden.

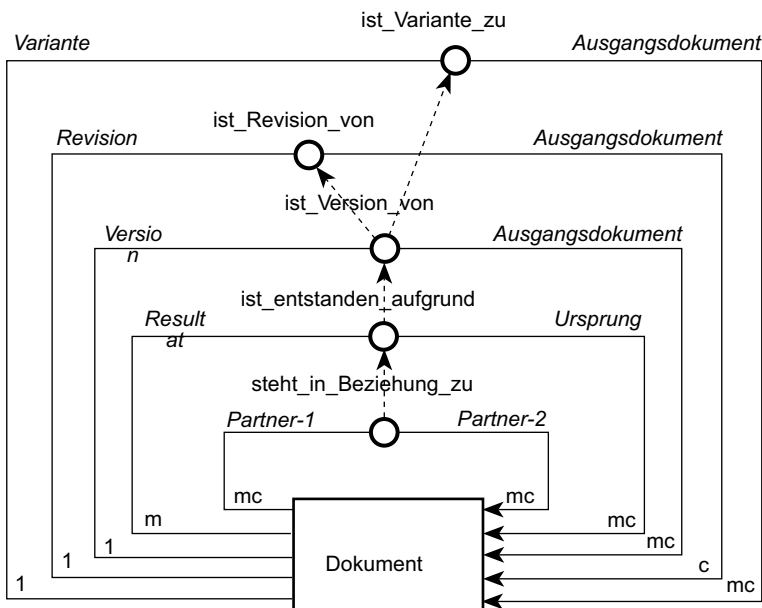


Abb. 3.7: Das DS-Schema

Die Beziehungsart *ist_entstanden_aufgrund* (Dokument, Dokument)

Sie ist eine Spezialisierung der Beziehungsart *steht_in_Beziehung_zu* (Dokument, Dokument) und modelliert, daß ein Dokument aufgrund eines anderen Dokument erzeugt wurde. Das erste Dokument in der Beziehung ist das neu entstandene Dokument. Das zweite Dokument ist der Ursprung, aufgrund dessen das neue Dokument erstellt wurde.

Konsistenzbedingungen:

- Ein Dokument kann der Ursprung für beliebig viele neue Dokumente sein. Zu einem neuen Dokument muß es jedoch mindestens einen Ursprung geben.
- Die Beziehungsart ist antisymmetrisch, irreflexiv und azyklisch.

Die Beziehungsart *ist_Version_von* (Dokument, Dokument)

Sie ist die allgemeine Beziehungsart, um die Entwicklungsgeschichte der Dokumente zu modellieren und eine Spezialisierung der Beziehungsart *ist_entstanden_aufgrund* (Dokument, Dokument). Sie modelliert, daß ein Dokument aufgrund von Veränderungen aus einem anderen Dokument

hervorgegangen ist. Das erste Dokument ist die neu erstellte Version, das zweite ist das Ausgangsdokument.

Konsistenzbedingungen:

- Zu einem Ausgangsdokument kann es beliebig viele Versionen geben. Eine Version ist jedoch aus genau einem Ausgangsdokument hervorgegangen.

Die Beziehungsart *ist_Revision_von (Dokument, Dokument)*

Sie ist eine Spezialisierung der Beziehungsart *ist_Version_von (Dokument, Dokument)* und modelliert, daß ein Dokument eine Weiterentwicklung des Ausgangsdokuments ist, das anschließend dieses ersetzt.

Konsistenzbedingungen:

- Zu einem Ausgangsdokument gibt es keine oder genau eine Revision; eine Revision ist genau aus einem Ausgangsdokument hervorgegangen.

Die Beziehungsart *ist_Variante_zu (Dokument, Dokument)*

Diese Beziehungsart ist eine weitere Spezialisierung der Beziehungsart *ist_Version_von (Dokument, Dokument)* und modelliert, daß ein Dokument aus einem Ausgangsdokument durch Modifikationen hervorgegangen ist und als Alternative dazu verwendet werden kann.

Konsistenzbedingungen:

- Zu einem Ausgangsdokument kann es beliebig viele Varianten geben; eine Variante ist genau aus einem Ausgangsdokument hervorgegangen.

3.6 Die Architektur der Dokumente

In diesem Abschnitt wird ein Architekturbegriff eingeführt, der auf Dokumenten basiert. Dieser wird anschließend auf die Architektur der Software übertragen.

3.6.1 Ein allgemeiner Architekturbegriff

Eine *Architektur* ist eine minimale Menge von miteinander in Beziehung stehenden Dokumenten, die durch ein *Architekturkriterium* definiert ist. Das Architekturkriterium muß für jede Architektur angegeben werden. Die Dokumente einer Architektur werden *Architekturdokumente* genannt.

Die in der Definition des Architekturbegriffs geforderte Eigenschaft „minimal“ wird folgendermaßen definiert:

- *Minimalität* bedeutet, daß das Architekturkriterium nicht mehr erfüllt ist, wenn ein Dokument aus der Architektur entfernt wird.

Abbildung 3.8 zeigt ein Beispiel für eine Architektur in diesem Sinn.

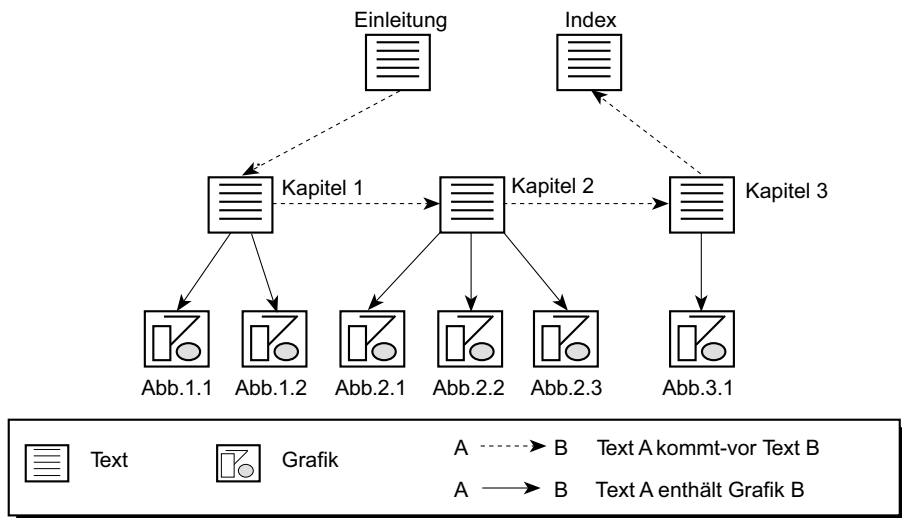


Abb. 3.8: Beispiel für eine Architektur

Diese Architektur besteht aus Text- und Grafikdokumenten sowie deren Beziehungen. Das Architekturkriterium dazu könnte folgendermaßen lauten: Zur Architektur gehören die Text- und Grafikdokumente der gesamten Dokumentenmenge sowie deren Beziehungen, die notwendig sind, um das Benutzungshandbuch zusammenzustellen.

3.6.2 Software-Architektur und Software-Architektur-dokumente

Der allgemeine Architekturbegriff läßt sich auf die Dokumente der Software-Entwicklung anwenden. Dies wird im folgenden diskutiert.

Ist die Software-Entwicklung abgeschlossen, werden nicht alle erstellten Dokumente benötigt, um das Zielprogramm verstehen, einsetzen und warten zu können. Zu diesen Dokumenten zählen möglicherweise Inter-

viewaufzeichnungen zwischen Kunde und Analytikern, Kostenpläne oder Prototyprealisierungen.

Aus der gesamten Dokumentenmenge kann somit eine Teilmenge isoliert werden, die folgendes Kriterium erfüllt:

Alle Dokumente dieser Menge sind für das Verständnis, den Einsatz und die Wartung des Zielprogramms notwendig.

Dieses wird als *Software-Architekturkriterium* bezeichnet. Die Teilmenge der Dokumente, die das Software-Architekturkriterium erfüllt, bildet die *Software-Architektur*. Die entsprechenden Dokumente sind die *Software-Architekturdokumente*.

Das Ziel jeder Software-Entwicklung besteht darin, eine Software-Architektur zu erstellen. Dafür werden aber immer auch Dokumente benötigt, die nicht zur Architektur gehören, jedoch für deren Entwicklung notwendig sind.

3.6.3 Die Software-Entwicklung und der Bau eines Gebäudes

Die seit langem beim Bau eines Gebäudes übliche, akzeptierte und bewährte Vorgehensweise kann als Vorbild für die Software-Entwicklung dienen.

Bevor ein Gebäude erstellt wird, entwirft ein Architekt ein Modell oder mehrere Modelle des Gebäudes, aus denen der Bauherr auswählen kann. Diese Modelle sind in der Regel Pläne und Zeichnungen. Bei einem größeren Bauvorhaben wird weiterhin auch ein Modell der Baustelle angefertigt. Das Modell der Baustelle zeigt beispielsweise, wo Baumaterialien gelagert oder wo die Büros der Bauleitung errichtet werden. Die Modellierung der Baustelle ist ausschließlich für den Konstruktionsprozeß des Gebäudes notwendig. Alle erstellten Modelle, der Plan des Gebäudes, der Plan der Baustelle und möglicherweise auch ein gegenständliches Modell des Gebäudes, das dessen äußere Form zeigt, können von allen Beteiligten betrachtet und bewertet werden. Gewünschte Änderungen können schnell und kostengünstig in das Modell eingearbeitet werden. Erst wenn alle Beteiligten die Pläne akzeptiert haben, wird mit der Realisierung der eigentlichen Architektur begonnen.

Diese Vorgehensweise ist auf die Software-Entwicklung übertragbar, da es für ein großes Software-System nicht möglich ist, auf Anhieb eine „geeignete“ Software-Architektur zu erstellen. Es ist also sinnvoll, zuerst ein

Modell oder mehrere alternative Modelle anzufertigen. Diese Modelle können bewertet werden, und es kann das Modell ausgewählt werden, das anschließend als Vorbild für die Konstruktion der Architektur verwendet wird. Dadurch wird erreicht, daß alle am Entwicklungsprozeß beteiligten Personengruppen vorab einen Eindruck von der geplanten Software erhalten und ihre Anforderungen und Wünsche einbringen können.

Bei der Software-Entwicklung sind die Architekturdokumente die wesentlichen und tragenden Dokumente. Die Dokumente, die nicht zur Software-Architektur gehören, können mit den Objekten verglichen werden, die auf der Baustelle eines Gebäudes existieren: Sie sind unbedingt für die Konstruktion notwendig, werden aber nicht mehr gebraucht, wenn der Konstruktionsprozeß abgeschlossen ist.

Software-Architekturdokumente sind dadurch charakterisiert, daß sie

- zu anderen Software-Architekturdokumenten konkrete *semantische* Beziehungen haben (z.B. Modul_1 benutzt Modul_2) und daß sie
- in einer jeweils spezifischen Art *vollständig* sind (z.B. Modul_1 erfüllt die Anforderungen seiner Spezifikation, ist übersetzbar und getestet).

3.6.4 Architekturen bei der Software-Entwicklung

In jeder Software-Architektur treten weitere Architekturen auf. Dazu gehört z.B. die Architektur der Spezifikationsdokumente. Auch bei der Architektur von Gebäuden gilt diese Aussage. So kann dort z.B. die Architektur der Anlagen identifiziert werden, die zur elektrischen Versorgung des Gebäudes dienen.

In jeder Software-Architektur existiert die *Programmarchitektur*. Sie wird von den Code-Dokumenten der Software-Architektur gebildet, die notwendig sind, um das lauffähige Zielprogramm zu erstellen.

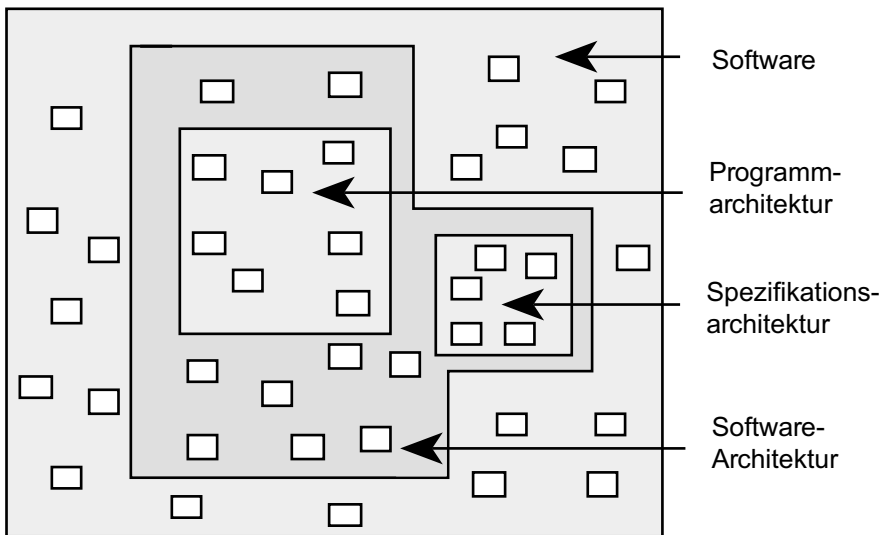


Abb. 3.9: Zusammenhang zwischen Software und darin auftretenden Architekturen

Abbildung 3.9 zeigt schematisch den Zusammenhang zwischen Software, Software-Architektur und Architekturen, die darin auftreten können. Die weißen Rechtecke symbolisieren einzelne Dokumente.

3.7 Eine Klassifikation für Software-Modellierungsansätze

Im folgenden wird eine Klassifikation für Software-Modellierungsansätze vorgestellt, die zum Software-Architektur-Prototyping führt. Diese Klassifikation beruht auf zwei Fragestellungen:

- Welcher Ausschnitt der Software wird betrachtet?
- Wie werden die Dokumente abgebildet, die das Zielprogramm modellieren?

Die Elemente eines Software-Modells sind nach dem vorher beschriebenen Ansatz die Dokumente und deren Beziehungen. Auf dieser Basis und mit dem eingeführten Architekturbegriff kann die in der folgenden Abbildung dargestellte Klassifikation für Software-Modellierungsansätze angegeben werden. Die Klassifikation definiert eine Spezialisierungs-Hierarchie zwischen den einzelnen Software-Modellierungsansätzen.

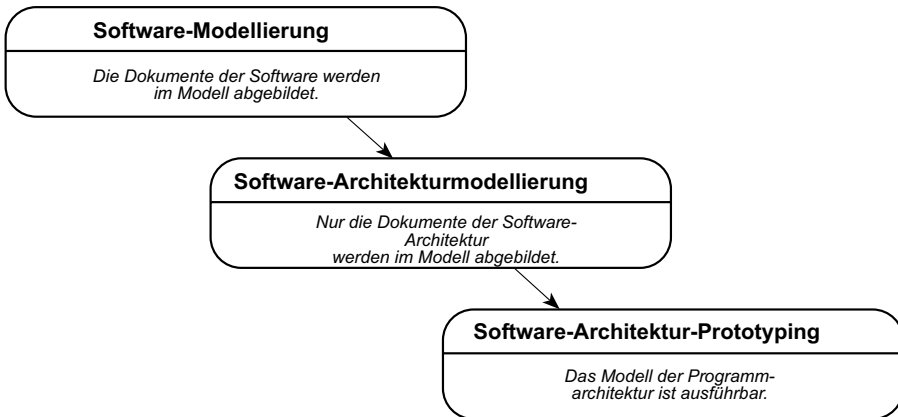


Abb. 3.10: Klassifikation von Software-Modellierungsansätzen

Software-Modellierung

ist der Prozeß, in dem ein Modell der Software erstellt wird, das die Dokumente zeigt.

Software-Architekturmodellierung

ist eine spezielle Art der Software-Modellierung, bei der nur ein Ausschnitt der Software, nämlich die Software-Architektur, modelliert wird.

Software-Architektur-Prototyping

ist eine spezielle Art der Software-Architekturmodellierung, bei der die Dokumente in Maschinencode übersetzt und ausgeführt werden können, die die Programmarchitektur modellieren. Die Dokumente der Programmarchitektur werden im weiteren als *Programmbausteine* bezeichnet.

Bei der Modellierung der Programmbausteine kann deren Eigenschaft, das sie übersetzbar sind und dadurch letztlich ausgeführt werden können, auf ihr Modell abgebildet werden oder nicht. Ein Software-Modellierungsansatz, bei dem die Modelle der Programmbausteine ausgeführt werden können, führt zu einem Modell des Zielprogramms, das ebenfalls ausgeführt werden kann. Das Modell des Zielprogramms ist nach der in Kapitel 2 gegebenen Definition ein Prototyp.

Software-Architektur-Prototyping ist somit der Prozeß, in dem ein Modell der Software-Architektur erstellt wird, wobei das Modell der Programmarchitektur ausgeführt werden kann. Der bei diesem Prozeß konstruierte Prototyp der Programmarchitektur wird als *Architekturprototyp* bezeichnet.

Damit ein Architekturprototyp ausgeführt werden kann, müssen die Programmbausteine mindestens rudimentär implementiert sein. Nach der Evolution des Architekturprototyps definiert dieser die Architektur des Zielprogramms.

Kapitel 4

Software-Architektur-Prototyping - Prinzip und Vorgehensweise

In Kapitel 3 wurde Software-Architektur-Prototyping als eine spezielle Art der Software-Modellierung definiert, bei der das Modell der Programmarchitektur ausführbar ist.

In diesem Kapitel werden Software-Architektur-Prototyping und die damit angestrebten Ziele eingehend erläutert. Darauf aufbauend wird ein Prozeßmodell vorgestellt, bei dem ein Architekturprototyp konstruiert und schrittweise in das Zielprogramm überführt wird. Es wird beschrieben, welche Werkzeugunterstützung dazu notwendig ist.

4.1 Warum Software-Architektur-Prototyping?

Software-Modelle werden erstellt, damit kritische Aspekte untersucht werden können. Bei der Software-Architekturmodellierung wird die Architektur, deren Güte entscheidenden Einfluß auf die Software-Qualität hat, am Modell untersucht. Dabei ist es hilfreich, wenn das Modell der Programmarchitektur nicht nur statisch untersucht, sondern auch ausgeführt werden kann. Das Verhalten des Programms kann dadurch besser beobachtet und analysiert werden.

Der Zusammenhang zwischen Programmarchitekturentwurf, dessen Ergebnis das Programmarchitekturmodell ist, und der Implementierung der Programmbausteine wird im nächsten Abschnitt diskutiert. Es zeigt sich, daß diese Tätigkeiten nicht getrennt voneinander betrachtet werden können.

4.1.1 Zusammenhang zwischen Programmarchitekturentwurf und Implementierung

Das Ergebnis des Entwurfs ist die software-technische Lösungsstruktur für die in der Spezifikation enthaltenen Anforderungen. Diese Tätigkeit

unterteilt Fairley (1985) in den *Architekturfentwurf* und in den *Feinentwurf* der Programmbausteine. Für den Architekturfentwurf wird in der Literatur häufig der von Kron und DeRemer (1976) eingeführte Begriff *Programming-in-the-Large* verwendet, um diese Tätigkeit von der eigentlichen Codierung, dem *Programming-in-the-Small*, abzugrenzen. Im folgenden wird diskutiert, wie diese beiden Tätigkeiten zusammenhängen.

Nagl (1990) benutzt den Begriff *diskretes Architekturparadigma*, um auszudrücken, daß alle Bausteine einer Programmarchitektur identifiziert werden können, ohne ihren internen Aufbau zu betrachten. Dies ist jedoch eine idealisierte Vorstellung, die nicht der tatsächlichen Vorgehensweise entspricht, weil scharf zwischen dem Architekturfentwurf und dem Feinentwurf der einzelnen Bausteine getrennt wird. Es ist nicht möglich, alle Bausteine und Beziehungen einer Programmarchitektur anzugeben, ohne in einem gewissen Umfang den internen Aufbau der Bausteine zu kennen.

Das folgende Beispiel stützt diese Aussage: Eine Beziehung „Modul A benutzt Modul B“ kann nur dann eingerichtet werden, wenn erkannt wurde, daß wenigstens eine Leistung des Moduls „B“ benötigt wird, um Modul „A“ zu realisieren. Dies kann aber nur dann festgestellt werden, wenn, wenigstens im Groben, die Realisierung des Bausteins „A“ durchdacht oder sogar durchgeführt wurde.

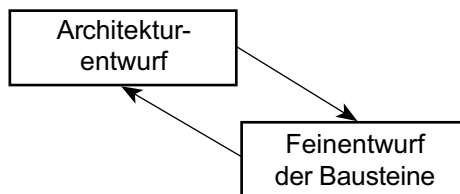


Abb. 4.1: Verzahnung der Tätigkeiten beim Entwurf

Eine Programmarchitektur wird entworfen, indem abwechselnd die Aktivitäten *Architekturfentwurf* und *Feinentwurf der Bausteine* durchgeführt werden, wobei mit dem Architekturfentwurf begonnen wird. Sind wesentliche Bausteine identifiziert, werden diese detailliert entworfen. Der Feinentwurf eines Bausteins führt jedoch häufig dazu, daß andere Bausteine der Architektur angepaßt werden müssen. Auf die Art und Weise entsteht sukzessiv die vollständige Programmarchitektur.

4.1.2 Das Prinzip des Software-Architektur-Prototyping

Software-Architektur-Prototyping berücksichtigt, daß die beiden Tätigkeiten beim Entwurf verzahnt durchgeführt werden. Die Bausteine einer Programmarchitektur werden deshalb im Modell so abgebildet, daß sie ausgeführt werden können. Dadurch kann besser festgestellt werden, wie ein Baustein in die gesamte Programmarchitektur eingebettet ist und welche Anforderungen seine Realisierung an die übrigen Bausteine stellt. Programmbausteine werden bis zu dem Detaillierungsgrad prototypisch realisiert, der die Beziehungen zu anderen Bausteinen erkennen läßt. Die Programmbausteine werden jedoch nicht in voller Funktionalität implementiert.

Der Schwerpunkt beim Software-Architektur-Prototyping liegt somit beim Architekturentwurf. Die prototypische Realisierung der Bausteine ist lediglich ein Mittel, um Informationen über die Architektur zu erhalten, die sonst nicht gewonnen werden können.

4.1.3 Charakterisierung des Software-Architektur-Prototyping

Jeder Modellierungsansatz kann dadurch charakterisiert werden, daß die in Kapitel 2 vorgestellten vier Grundfragen der Modellierung beantwortet werden. Dies gilt auch für das Software-Architektur-Prototyping.

Was ist das Original des Modells?

Das Original des Modells ist die Software-Architektur, wobei das Original noch nicht realisiert ist, jedoch virtuell existiert. Das Architekturmodell dient als Vorbild, um das Original herzustellen.

Zu welchem Zweck wird das Modell erstellt, welche Fragestellung soll geklärt werden?

Mit einem Software-Architekturmodell soll folgende Frage beantwortet werden: Wie sieht eine Software-Architektur aus, die als software-technische Lösungsstruktur für die gegebene Aufgabenstellung geeignet ist und die gestellten Qualitätsanforderungen erfüllt? Um diese Frage beantworten zu können, ist folgendes notwendig:

- Es müssen die Qualitätsanforderungen definiert werden, die die Architektur erfüllen soll.
- Der Architekturprototyp muß ausgeführt und bewertet werden.

Wer sind die Modellbenutzer?

Die Modellbenutzer sind die Entwerfer selbst, aber auch Fachleute auf dem Gebiet des Entwurfs, die selbst nicht an der Entwicklung beteiligt sind. Sie untersuchen die Güte der konstruierten Architektur. Da der Architekturprototyp auch rudimentär das Verhalten des Programms zeigt, kann auch der Kunde den Architekturprototyp gegen seine Erwartungen prüfen.

Wie lange soll das Modell benutzt werden?

Das Architekturmodell wird solange geprüft und modifiziert, bis es insgesamt als ausreichend bewertet wird, um als Vorbild für die Konstruktion des Originals verwendet zu werden. Die Informationen, die mit dem Architekturmodell gewonnen werden, müssen auf das Original, d.h. auf die Architektur des Zielsystems, übertragen werden. Die Möglichkeiten, die dazu bestehen, werden im nächsten Abschnitt diskutiert.

4.1.4 Der Übergang vom Architekturmodell zum Zielsystem

Ein Software-Architekturmodell wird immer als Vorbild für die Konstruktion des Zielsystems verwendet. Es muß deshalb diskutiert werden, wie die am Modell gewonnenen Informationen in die Konstruktion des Zielsystems einfließen können.

Für einen Übergang vom Architekturprototyp zur Architektur des Zielprogramms existieren die folgenden Alternativen:

- Der Architekturprototyp wird solange geprüft und modifiziert, bis er als Ganzes den gestellten Anforderungen entspricht. Das Modell ist anschließend die Entwurfsspezifikation, nach der das Zielprogramm erstellt wird. Die mit dem Architekturprototyp gesammelten Erfahrungen und Informationen fließen über diesen Weg in die Konstruktion des Zielprogramms ein.
- Der Architekturprototyp wird schrittweise erstellt und geprüft. Stellt sich bei der Prüfung des Prototyps heraus, daß er als Ganzes zwar noch nicht allen Anforderungen entspricht, jedoch Programmbausteine enthält, die ein stabiles Verhalten zeigen, so können diese Bausteine bereits in die Architektur des Zielprogramms übertragen werden.

Im nächsten Abschnitt wird ein Prozeßmodell eingeführt, das die zweite Vorgehensweise bei Übergang zum Zielprogramm unterstützt.

4.2 PDSC – Der schrittweise Übergang vom Architekturprototyp zum Zielprogramm

PDSC (**P**rogram **D**evelopment by **S**teppwise **C**ompletion) ist eine Methode für das Software-Architektur-Prototyping, bei dem das Zielprogramm ausgehend von einem Architekturprototyp schrittweise entwickelt wird.

PDSC wird erstmals in Ludewig et. al. (1987) beschrieben und geht von den folgenden Annahmen aus:

- Eine scharfe Trennung zwischen der Spezifikations- und der Entwurfsphase ist unrealistisch und unzweckmäßig, weil sich die Anforderungen zu einem großen Teil erst durch Entwurfsentscheidungen ergeben (Swartout, Balzer, 1982; Ludewig, 1982)
- Software-Entwicklung ist ein Prozeß, bei dem solange Unvollständigkeiten beseitigt werden, bis die Software fertig gestellt ist und ausgeliefert werden kann.

Der zweite Aspekt wird nachfolgend näher erläutert.

4.2.1 Unvollständigkeiten bei der Software-Entwicklung

Software-Entwicklung ist ein Prozeß, in dem schrittweise, ausgehend von einem unvollständigen Zustand, Unvollständigkeiten beseitigt werden, bis die fertige Software entstanden ist.

Die größte Unvollständigkeit liegt beim Projektstart vor. Zu diesem Zeitpunkt ist außer vagen Ideen und Wünschen, die durch eine Software verwirklicht werden sollen, nichts bekannt. Aber auch während der Entwicklung existieren Unvollständigkeiten, da weder alle Teile der Software identifiziert noch vollständig beschrieben werden können.

Die Aufgabe der Software-Entwickler besteht nun darin, diese Unvollständigkeit zu beseitigen, um zur fertigen Software zu gelangen.

Ludewig et. al (1985b) unterscheidet prinzipiell drei Arten der Unvollständigkeit:

1. *Fehlende Teile*

Aus Gründen der Übersichtlichkeit werden Teile, die zwar vorhanden sind, nicht betrachtet. Dadurch wird das Wesentliche herausgestellt.

2. *Unbestimmte Teile*

Es wird die Entscheidung offen gelassen, wie ein Teil genau aussehen wird.

Beispiel: Es ist zwar bekannt, daß ein Programmbaustein für die Datenhaltung gebraucht wird. Es ist jedoch noch offen, ob dies eine kommerziell erhältliche Datenbank oder eine eigene, spezielle Datenhaltungskomponente sein wird.

3. *Fehlende Detaillierung*

Die Einzelheiten eines Teils werden nicht dargestellt, weil diese nicht von Interesse oder nicht bekannt sind.

Beispiel: Für einen Programmbaustein ist noch nicht definiert, welche Operationen er zur Verfügung stellen wird.

Die Entwickler brauchen methodische und sprachliche Möglichkeiten, um Unvollständigkeiten *formulieren* und während der Entwicklung *verwalten* und *aflösen* zu können.

PDSC berücksichtigt diese Betrachtungsweise der Software-Entwicklung. Die Methode PDSC unterstützt die Entwickler dadurch, daß sie Software-Architektur-Prototyping in der Form vorsieht, bei der der erstellte Architekturprototyp schrittweise in das Zielprogramm überführt wird. Dabei werden nach und nach Unvollständigkeiten beseitigt.

In den nächsten Abschnitten wird PDSC vorgestellt. Es werden die folgenden Fragen diskutiert und beantwortet:

- Wie kann ein Architekturprototyp schrittweise vervollständigt werden?
- Welche Tätigkeiten sind dazu notwendig und welche Rollen können identifiziert werden?
- Welche Sprachen werden dazu benötigt und welche Anforderungen sind an diese zu stellen?
- Welche Voraussetzungen müssen erfüllt sein, damit ein Programmbaustein in einer Zielsprachenimplementierung verwendet werden kann?
- Welche Werkzeugunterstützung wird für PDSC benötigt?

4.2.2 Die schrittweise Vervollständigung des Architekturprototyps

Ein Architekturprototyp besteht anfänglich aus Programmbausteinen, die überwiegend in Prototypimplementierung vorliegen. Lediglich Programm-

bausteine, die bereits in Zielsprache realisiert sind und wiederverwendet werden, und Programmbausteine, die so einfach sind, daß sie direkt in der Zielsprache codiert werden können, werden nicht in einer Prototypimplementierung verwendet.

Der ausführbare Architekturprototyp kann geprüft und bewertet werden. Als Ergebnis dieser Prüfung wird der Architekturprototyp verändert und weiter detailliert, d.h. es werden Unvollständigkeiten beseitigt. Wenn sich bei der Prüfung des Architekturprototyps herausstellt, daß Teile ein stabiles Verhalten zeigen, werden diese in der Zielsprache realisiert. Nachdem sie getestet sind, werden sie wieder in den Architekturprototyp integriert. Dabei muß jedoch gewährleistet sein, daß der Architekturprototyp auch dann ausgeführt werden kann, wenn er aus Programmbausteinen besteht, die teils in einer Prototyp-, teils in einer Zielsprachenimplementierung vorliegen.

Im Laufe der Entwicklung verringert sich der Anteil der Programmbausteine, die prototypisch implementiert sind, in dem Maß, wie der Anteil der in Zielsprache implementierten Programmbausteine zunimmt. Aus der Prototypimplementierung eines Programmbausteins wird nur dessen Einbindung in die Architektur, die in seiner Schnittstelle beschrieben ist, in die Zielsprachenimplementierung übernommen. Die prototypische Implementierung wird nicht verwendet.

Bis zum letzten Vervollständigungsschritt findet die Entwicklung auf zwei Sprachebenen statt: auf der Ebene der Prototyping-Sprache und auf der Ebene der Zielsprache. Die folgenden Abbildungen zeigen diese Entwicklung.

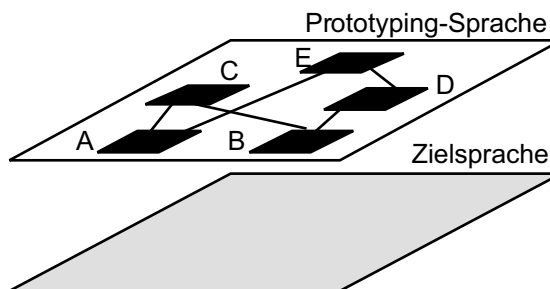


Abb. 4.2 a: Die schrittweise Vervollständigung (Anfangszustand)

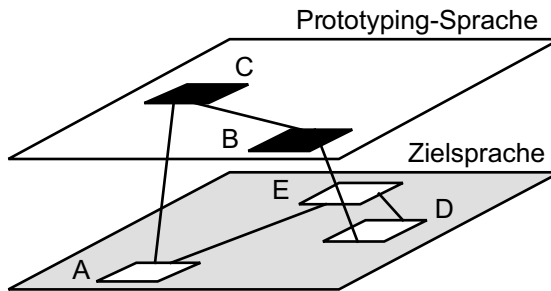


Abb. 4.2 b: Die schrittweise Vervollständigung (Zwischenzustand)

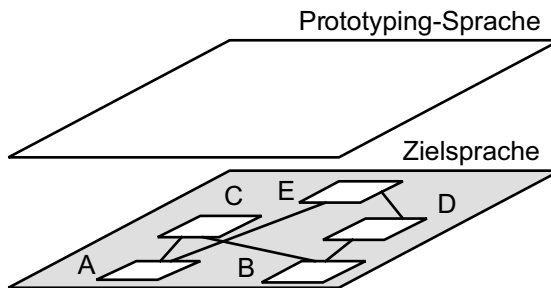


Abb. 4.2 c: Die schrittweise Vervollständigung (Endzustand)

Im dargestellten Beispiel sind zu Beginn der Entwicklung alle Programmbausteine in der Prototyping-Sprache realisiert (Abb. 4.2 a). Während der Entwicklung werden die bereits stabilen Programmbausteine (im Beispiel „A“, „E“ und „D“) in der Zielsprache codiert (Abb. 4.2 b). Nachdem sie in den Architekturprototyp integriert sind, bilden sie zusammen mit den Programmbausteinen „B“ und „C“, die noch in der Prototypimplementierung vorliegen, den Architekturprototyp. Die in der Zielsprache codierten Programmbausteine werden dabei genau so in den Architekturprototyp eingepaßt wie ihre Prototypimplementierungen.

Das Zielprogramm ist dann vollständig realisiert, wenn alle Programmbausteine nicht mehr in ihrer Prototypimplementierung vorliegen, sondern durch ihre Zielsprachenimplementierung ersetzt worden sind (Abb. 4.2 c).

4.2.3 Tätigkeiten und Rollen bei PDSC

Bei der Software-Entwicklung mit PDSC müssen spezielle Tätigkeiten durchgeführt werden. Diesen Tätigkeiten lassen sich einzelne Rollen

zuordnen. Die folgende Abbildung zeigt die Tätigkeiten und die dabei produzierten Ergebnisse.

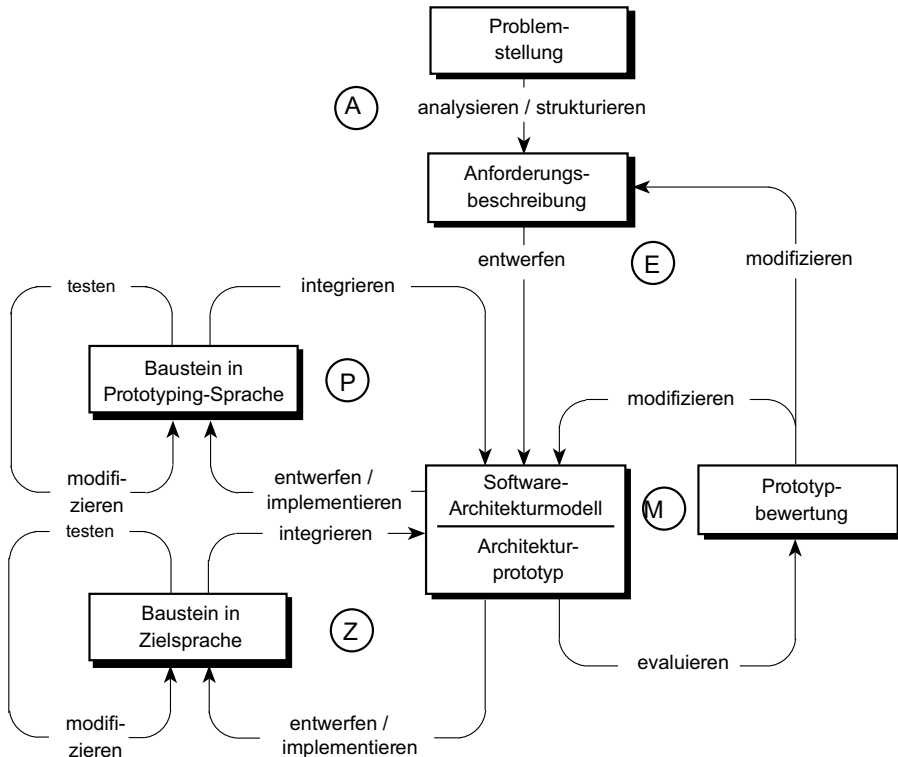


Abb. 4.3: Tätigkeiten bei PDSC

Die einzelnen Aktivitäten bei PDSC sind:

- A** Analysieren der Problemstellung und Strukturieren der Anforderungen zu einer informalen Anforderungsbeschreibung
- E** Entwerfen eines Software-Architekturmodells
- P** Implementieren der einzelnen Programmbausteine in der Prototyping-Sprache, Testen und Integrieren in den Architekturprototyp
- M** Evaluieren und Modifizieren des Architekturprototyps
- Z** Implementieren der einzelnen Programmbausteine in der Zielsprache, Testen und Integrieren in den Architekturprototyp

Natürlich beginnt die Arbeit mit den Aktivitäten A, E, P und M. Danach ist die Reihenfolge weitgehend beliebig. Durch die Prüfung des Architekturprototyps soll erreicht werden, daß jeder einzelne Programmbaustein nur einmal in der Zielsprache codiert werden muß. Dieses Ziel wird sich in kritischen Fällen nicht immer erreichen lassen.

Nachfolgend werden die Tätigkeiten einzeln diskutiert. Dabei werden jeweils Rollenbezeichnungen für die Personen angegeben, die die betreffenden Tätigkeiten ausführen. Eine bestimmte Person kann auch mehrere dieser Rollen innehaben.

A Analysieren der Problemstellung und Strukturieren der Anforderungen zu einer informalen Anforderungsbeschreibung

Am Anfang der Entwicklung steht die Analyse (*Analytiker*) der Problemstellung. Auf der Basis der Analyse wird das Problem in überschaubare Teile gegliedert (*Spezifizierer*). Die resultierende Anforderungsbeschreibung ist die Grundlage für den anschließenden Entwurf. PDSC bietet keine methodische Unterstützung für diese Tätigkeiten.

E Entwerfen eines Software-Architekturmodells

Auf der Basis der Ergebnisse von A wird ein Software-Architekturmodell entworfen, das die Struktur, die Bezeichner, den Zweck und die Beziehungen der Dokumente festlegt (*Entwerfer*). Das Software-Architekturmodell wird in einer Sprache formuliert, die als *Architekturbeschreibungssprache* bezeichnet wird. Sie bietet die im Software-Schema definierten Möglichkeiten zur Modellierung an.

P Implementieren der einzelnen Programmbausteine in der Prototyping-Sprache, Testen und Integrieren in den Architekturprototyp

Die einzelnen Programmbausteine werden in der Prototyping-Sprache implementiert (*Prototypprogrammierer*). Damit entsteht erstmals ein ablauffähiges Programm. Soweit Programmbausteine bereits in der Zielsprache vorliegen oder mit geringem Aufwand in Zielsprache codiert werden können, entfällt natürlich die Implementierung als Prototyp.

Es sei darauf hingewiesen, daß Entwerfer und Prototypprogrammierer, falls sie nicht ein und dieselbe Person sind, zusammen im Entwurfsteam arbeiten müssen, weil Anforderungen an andere Bausteine identifiziert werden, wenn ein Programmbaustein prototypisch implementiert wird. Diese können das Architekturmodell als Ganzes verändern.

Durch die Eigenschaften der Prototyping-Sprache ist gewährleistet, daß der Prototypprogrammierer mit geringem Aufwand und ohne starke Einschränkung durch die „Sicherheitsgurte“ einer Zielsprache, also Typüberwachung und Schnittstellenprüfung, zu Ergebnissen gelangt.

M Evaluieren und Modifizieren des Architekturprototyps

Sobald der Architekturprototyp ausgeführt werden kann, kann er erprobt und geprüft werden (*Prototypprüfer*). Dazu müssen nicht alle Programmbausteine, aber ein zentraler Kern in einer Prototypimplementierung vorliegen. Folgende Prüfungen können am Architekturprototyp durchgeführt werden

- Analytiker und Spezifizierer prüfen gegen die Anforderungen, die in der Anforderungsbeschreibung formuliert sind.
- Entwerfer und Fachleute des Entwurfs prüfen gegen die speziellen Anforderungen, die die Architektur erfüllen muß. Diese sind ebenfalls in der Anforderungsbeschreibung fixiert.
- Der Kunde prüft gegen seine (oft nicht formulierten) Erwartungen.
- Fachleute für spezielle Aspekte (z.B. Effizienz, Wartbarkeit) prüfen gegen die betreffenden (meist nichtfunktionalen) Anforderungen.

Der Architekturprototyp wird jeweils auf der Grundlage der vollständigsten Konfiguration evaluiert, die zu einem Zeitpunkt möglich ist. In einer Konfiguration werden jüngere Revisionen der Programmbausteine den älteren vorgezogen, solche in Zielsprache denen in Prototyping-Sprache.

Diese Prüfungen haben primär zum Ziel, Mängel bezüglich der Anforderungsspezifikation und der Programmarchitektur zu erkennen. Mängel können in diesem Stadium mit wesentlich weniger Aufwand beseitigt werden, als nach der Codierung in Zielsprache.

Das Ergebnis dieser Erprobung ist eine *Prototypbewertung*. Die Erprobung endet, wenn das Ergebnis insgesamt akzeptabel ist. Andernfalls müssen die Tätigkeiten E, P und M erneut ausgeführt werden, im ungünstigsten Fall auch A.

Z Implementieren der einzelnen Programmbausteine in der Zielsprache, Testen und Integrieren in den Architekturprototyp

Ist ein Programmbaustein mit ausreichender Sicherheit stabil, kann er in der Zielsprache codiert werden (*Zielsprachenprogrammierer*). Die

Implementierung wird schon früher stattfinden, wenn sie zur Klärung wichtiger Fragen notwendig ist, beispielsweise, wenn die Antwortzeit eines Programmbausteins eine kritische Größe darstellt.

4.3 Übergang von Prototyp- zur Zielsprachenimplementierung

Da der Architekturprototyp schrittweise zum Zielprogramm entwickelt wird, müssen bei dessen Prüfung speziell die folgenden zwei Fragen beantwortet werden:

- Welche Bausteine können durch eine Zielsprachenimplementierung ersetzt werden?
- In welcher Reihenfolge muß dies geschehen?

4.3.1 Wann geschieht der Übergang?

Um diese Frage beantworten zu können, muß definiert sein, welche Anforderungen ein Programmbaustein erfüllen muß, damit seine Prototypdurch eine Zielsprachenimplementierung ersetzt werden kann. Wird der Übergang zu früh durchgeführt, muß die Zielsprachenimplementierung, falls Änderungen erforderlich sind, mehrmals angepaßt werden. Dies soll jedoch vermieden werden.

Folgende Anforderungen müssen erfüllt sein, wenn die Prototypimplementierung eines Programmbausteins durch die Zielsprachenimplementierung ersetzt werden soll:

- Die Schnittstelle des Programmbausteins muß sich in den vorhergehenden Prüfungs- und Modifikationszyklen stabilisiert haben. Dies bedeutet, daß der Programmbaustein selbst keine weiteren Anforderungen an andere Bausteine stellt und daß die übrigen Programmbausteine keine Anforderungen an ihn stellen, die er nicht bereits erfüllt.
- Der Programmbaustein muß die Qualitätsanforderungen erfüllen, die in der Anforderungsbeschreibung für alle Bausteine formuliert wurden. So kann beispielsweise die Anzahl der Beziehungen zu anderen Programmbausteinen (z.B. nicht mehr als acht benutzt-Beziehungen pro Baustein) oder die Anzahl der exportierten Operationen pro Baustein (z.B. nicht mehr als 20 Operationen) begrenzt sein.

Der Entwerfer muß letztlich aufgrund seiner Erfahrung entscheiden, wann ein Programmbaustein in einer Zielsprachenimplementierung in den Architekturprototyp integriert werden kann. Ein allgemeingültiges Entscheidungskriterium kann nicht angegeben werden.

4.3.2 Die Reihenfolge beim Übergang

Sollen mehrere Programmbausteine in ihrer Zielsprachenimplementierung in den Architekturprototyp integriert werden, so muß geklärt werden, in welcher Reihenfolge dies geschehen kann.

Die folgende Regel legt die Reihenfolge fest:

Ein in Zielsprache implementierter Programmbaustein kann dann und nur dann in den Architekturprototyp integriert werden, wenn alle Programmbausteine, die von ihm verwendet werden, ebenfalls in einer Zielsprachenimplementierung vorliegen.

Die Reihenfolge beim Übergang der einzelnen Programmbausteine kann somit mit dem Begriff *bottom-up* charakterisiert werden.

Die „bottom-up-Reihenfolge“ bei der Integration ist deshalb notwendig, damit ein Programmbaustein, der bereits in Zielsprache implementiert ist, nicht jedesmal modifiziert werden muß, wenn sich die Schnittstellen von verwendeten Bausteinen, die noch in Prototypimplementierung vorliegen, im Zuge der Evolution des Architekturprototyps verändern.

Ein Programmbaustein soll im Normalfall möglichst nur einmal in der Zielsprache implementiert, getestet und integriert werden.

Der strenge „bottom-up-Übergang“ führt zu folgenden Konsequenzen:

- Programmbausteine, die sich wechselseitig verwenden, müssen gleichzeitig in eine Zielsprachenimplementierung überführt werden.
- Die Laufzeitumgebung der Prototyping-Sprache hat solange die Kontrolle über den ausführbaren Architekturprototyp, bis der letzte Programmbaustein (das „Hauptprogramm“) in Zielsprachenimplementierung vorliegt. Dann ist aus dem Architekturprototyp das fertige Zielprogramm geworden.

4.4 Sprachen für PDSC

Für PDSC werden neben einer natürlichen Sprache, die für die informalen Texte verwendet wird, folgende Sprachen benötigt:

1. Eine Sprache, in der das Software-Architekturmodell formuliert werden kann. Dies ist die *Architekturbeschreibungssprache*. Mit ihr können Architekturmodelle gemäß dem zugrundeliegenden Software-Schema formuliert werden. Die Wahl des Software-Schemas bestimmt die *Architekturbeschreibungssprache*.
2. Eine Sprache, in der die Prototypimplementierungen der Programmbausteine formuliert werden. Dieses ist die *Prototyping-Sprache*. Mit ihr wird das dynamische Verhalten der Programmbausteine beschrieben.
3. Eine Sprache, in der die Programmbausteine für das Zielprogramm codiert werden; die *Zielsprache*.

4.4.1 Anforderungen an PDSC-Sprachen

Die Architekturbeschreibungssprache muß folgenden Anforderungen genügen:

- Es muß möglich sein, die Bausteine und Beziehungen, die das zugrundeliegende Software-Schema vorsieht, zu formulieren.
- Es müssen unvollständige und vage Beschreibungen erlaubt sein.
- Die Sprache soll zum überwiegenden Teil in einer grafischen Syntax vorliegen, die übersichtlich und maschinell verarbeitbar ist.

Standish (1982) gibt eine ausführliche Liste von Anforderungen an, die an eine Prototyping-Sprache gestellt werden. Im Rahmen des Einsatzes bei PDSC gilt:

- Sie muß interpretierbar sein, damit die langwierigen „Edit-Compile-Run“-Zyklen vermieden werden.
- Sie muß erlauben, daß unvollständige Programme ausgeführt werden können.
- Sie darf keinen Deklarationszwang für Bezeichner besitzen.
- Sie muß die schnelle Realisierung der Programmbausteine gestatten.

Für die Zielsprache gilt:

- Die Sprache muß die Wartung des Zielprogramms durch eine gut lesbare Notation und durch Deklarationszwang für die Bezeichner erleichtern.
- Sie muß eine strenge Typprüfung besitzen.
- Sie muß das Modulkonzept anbieten und die separate Übersetzung gestatten.
- Sie muß in effizienten Maschinencode übersetzbar sein.

4.4.2 Beziehungen zwischen Prototyping- und Zielsprache

PDSC sieht vor, daß die Prototypimplementierung eines Programmbausteins durch eine Zielsprachenimplementierung ersetzt wird. Damit dies einfach möglich ist, müssen Prototyping- und Zielsprache gewisse Voraussetzungen erfüllen. Diese werden im weiteren diskutiert.

Bei der Wahl der Prototyping- und Zielsprache sind grundsätzlich folgende Alternativen denkbar:

1. Prototyping- und Zielsprache sind identisch

Der Übergang eines Programmbausteins kann im wesentlichen dadurch erfolgen, daß die Prototypimplementierung verfeinert, teilweise ersetzt und erweitert wird. Die Schnittstelle eines Programmbausteins kann direkt für die Zielsprachenimplementierung verwendet werden, weil beide Sprachen identisch sind.

Da die Anforderungen an Prototyping- und Zielsprache allerdings stark voneinander abweichen, ist eine Sprache für beide Zwecke nur eine theoretisch aber keine praktisch relevante Alternative. Eine Sprache, die für das Prototyping und für die endgültige Implementierung gleich gut geeignet ist, existiert zur Zeit nicht. Diese Sprache müßte sowohl in einer permissiven Form (ohne Typprüfung, interpretiert) und in einer strengen Form (Typprüfung, compiliert) einzusetzen sein.

2. Die beiden Sprachen sind verschieden, gehören jedoch zur selben Klasse von Programmiersprachen

Die in der Prototyping-Sprache formulierte Schnittstelle eines Programmbausteins kann in eine entsprechende Zielsprachenformulierung transformiert werden, da die Zielsprache alle Konzepte der Prototyping-Sprache

zur Verfügung stellt. Der eigentliche Prototyp-Code eines Programmbausteins wird nicht verwendet und weggeworfen.

Bei dieser Sprachenkonstellation kann es jedoch vorkommen, daß Konzepte der Zielsprache nicht verwendet werden, weil die Prototyping-Sprache diese nicht bereitstellt. Wird beispielsweise Smalltalk-80 als Prototyping-Sprache und C++ als Zielsprache gewählt, so wird das von C++ zur Verfügung gestellte Konzept der Mehrfachvererbung nicht verwendet.

3. Prototyping- und Zielsprache sind verschieden und gehören unterschiedlichen Klassen von Programmiersprachen an

Diese Alternative scheidet bei einer Sprachenauswahl aus. Ein Übergang zwischen den Sprachen ist, wenn überhaupt, nur schwer möglich, da unterschiedliche Sprachkonzepte aufeinander abgebildet werden müssen.

Ein Beispiel für diese Art der Sprachenauswahl ist Smalltalk-80 als Prototyping-Sprache und Ada als Zielsprache. Die Vererbung, ein wesentliches Konzept in Smalltalk-80, kann nicht sinnvoll mit den Mitteln der Sprache Ada nachgebildet werden.

Aus den Überlegungen zur Sprachenauswahl folgt, daß Prototyping- und Zielsprache im Sinn der zweiten Alternative gewählt werden müssen.

4.5 Die Entwicklungsgeschichte der Dokumente bei PDSC

In Abschnitt 3.4 wird erläutert, wie die Entwicklungsgeschichte der Dokumente in allgemeiner Form modelliert werden kann. Im folgenden wird diskutiert, welchen Einfluß PDSC, und dabei speziell der schrittweise Übergang von der Prototyp- zur Zielsprachenimplementierung der Programmbausteine, auf die Entwicklung der Dokumente hat.

Beim Software-Architektur-Prototyping mit PDSC entwickeln sich die Software-Architekturdokumente während des Modellierungsprozesses, also in Revisionen. Da ein wesentliches Ziel dieser Vorgehensweise darin besteht, verschiedene Lösungsalternativen zu erstellen, werden ebenfalls Varianten entwickelt, die erprobt werden. Dies gilt im besonderen für die Programmbausteine der Architektur. Revisionen und Varianten werden aber nicht nur für die Modelle, das sind die Prototypimplementierungen, sondern auch für die Originale, das sind die Zielsprachenimplementierungen, erstellt. Bei der Revisionen- und Variantenbildung einer Ziel-

sprachenimplementierung muß jedoch beachtet werden, daß die Einbettung des Bausteins in die Architektur nicht verändert wird.

Das folgende Beispiel, erläutert die spezielle Revisionen- und Variantenbildung bei PDSC.

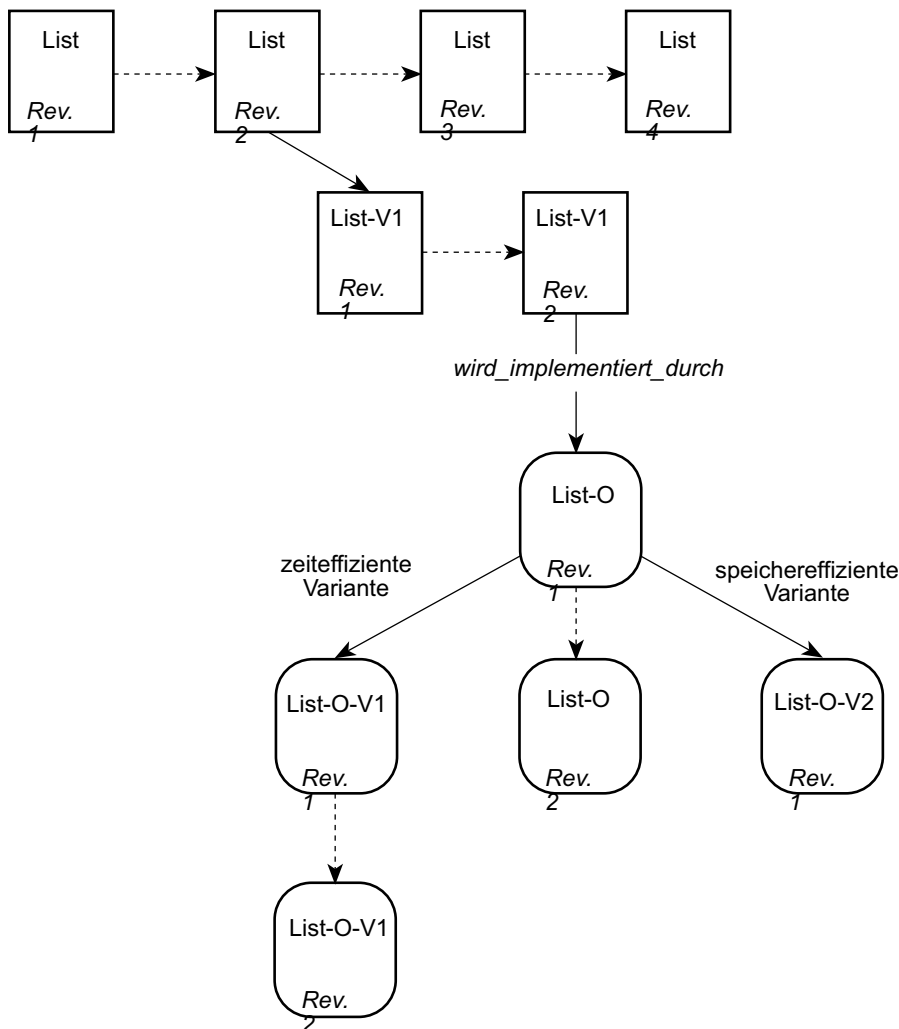


Abb. 4.4: Revisionen- und Variantenbildung bei PDSC

Es soll ein Programmbaustein entwickelt werden, mit dem Listen beliebiger Datenelemente gespeichert und verwaltet werden sollen. Dazu wird ein Modell (List) erstellt, das die prozedurale Schnittstelle zeigt, die der Baustein zur Verfügung stellen soll. Wie die Liste gespeichert und die Datenelemente verwaltet werden, ist nicht von Interesse. Das Modell wird in mehreren Revisionen weiterentwickelt. Dabei wird entschieden, auf der Grundlage von Revision 2 eine Variante zu entwickeln. Von dieser werden wiederum zwei Revisionen erstellt.

Nachdem beide alternative Bausteine („List“ in der Revision 4 und „List-V1“ in der Revision 2) bewertet sind, wird entschieden, die in „List-V1“ modellierte Schnittstelle als Vorbild für die Entwicklung des Programmbausteins in der Zielsprache zu verwenden (List-O). Nachdem dieser implementiert ist, werden zwei Varianten davon erstellt, die dieselbe Schnittstelle haben: eine, die eine möglichst schnelle Verwaltung realisiert (List-O-V1), und eine, die möglichst speichereffizient verwaltet (List-O-V2).

Die Revisionen- und Variantenbildung, sowohl bei der Prototyp- als auch bei der Zielsprachenimplementierung der Programmbausteine, muß berücksichtigt werden, wenn ein Architekturprototyp konfiguriert wird. Bei der Konfiguration des Architekturprototyps muß für jeden Programmbaustein entschieden werden, welche Revision bzw. welche Variante zur aktuellen Konfiguration gehört. Bei PDSC wird davon ausgegangen, daß die aktuelle Revision älteren Revisionen und eine Zielsprachenimplementierung (das Original) einer Prototypimplementierung (dem Modell) vorgezogen wird.

4.6 Konzeption einer Werkstatt für PDSC

PDSC kann nur durchgeführt werden, wenn die dabei anfallenden Tätigkeiten durch geeignete Werkzeuge unterstützt werden. Diese Werkzeuge werden in Form einer Werkstatt zusammengefaßt. Ihr Aufbau wird in diesem Abschnitt beschrieben und es wird gezeigt, welche Werkzeuge welche Tätigkeiten unterstützen.

4.6.1 Die Datenhaltung

Die bei einer Software-Entwicklung produzierte Informationsmenge kann mit einer Datenbank verwaltet werden. Diese muß spezielle Anforderungen erfüllen, die der Software-Entwicklungsprozeß stellt (z.B. Revisionen- und Variantenverwaltung, lange Transaktionen). Diese Datenbanken werden in Glinz (1985) als *Software-Engineering-Datenbanken* (SEDB) bezeichnet.

Matheis (1990) diskutiert in diesem Zusammenhang folgende Fragestellungen:

- Welche Vorteile entstehen für die Software-Entwicklung, wenn eine SEDB für die Verwaltung der Informationen eingesetzt wird?
- Welche Arten von Informationen entstehen im Laufe einer Software-Entwicklung?
- Welche Anforderungen sind an eine SEDB zu stellen?
- Welche Konzepte muß eine SEDB zur Verfügung stellen?

Das Herz einer Werkstatt für PDSC bildet eine SEDB. Diese Datenbank verwaltet alle Informationen, die entstehen, und unterstützt die Konfiguration des Architekturprototyps.

Der Kern eines SEDB-Datenbankschemas für PDSC kann aus dem verwendeten Software-Schema gewonnen werden. Da dieses in einer ER-Notation formuliert ist, ist eine Abbildung auf das Schema einer ER-Datenbank sehr einfach. Objektorientierte Datenbanken sind nach Ahmed et al. (1992) ebenfalls für die Informationsverwaltung bei der Software-Entwicklung geeignet. Das Software-Schema muß dazu auf das Schema einer objektorientierten Datenbank abgebildet werden; dies ist möglich.

4.6.2 Die Werkzeuge einer Werkstatt für PDSC

Die Tätigkeiten, die speziell bei PDSC anfallen und in Abschnitt 4.2.3 beschrieben sind, können durch Werkzeuge unterstützt werden. Nachfolgend werden den einzelnen Tätigkeiten oder Tätigkeitsgruppen Werkzeuge zugeordnet.

A Analysieren der Problemstellung und Strukturieren der Anforderungen zu einer informalen Anforderungsbeschreibung

PDSC bietet keine spezielle Werkzeugunterstützung für diese Tätigkeit an. Da die Informationen zu diesem Zeitpunkt kaum Struktur aufweisen, ist ein *Texteditor* das geeignete Werkzeug. Dieser wird auch verwendet, um alle informale Texte zu erstellen.

E Entwerfen eines Software-Architekturmodells

Das Architekturmodell wird in einer Architekturbeschreibungssprache formuliert. Dazu wird ein *Architektureditor* benötigt, mit dem die Architekturmodelle in vorwiegend grafischer Notation formuliert werden

können. Dieses Werkzeug muß es erlauben, die Bausteine und Beziehungen einer Architektur zu definieren und muß unterschiedliche Sichten auf die Architektur erlauben. Der Architektureditor übergibt die erstellten Architekturmodelle der SEDB.

P Implementieren der einzelnen Programmbausteine in der Prototyping-Sprache, Testen und Integrieren in den Architekturprototyp

Aus dem Modell eines Programmbausteins kann, bevor er prototypisch implementiert wird, ein Rahmen erzeugt werden, der der Syntax der Prototyping-Sprache entspricht. Dies kann automatisch durch ein Werkzeug geschehen (*PT-Rahmengenerator*).

Da es sich bei der Prototyping-Sprache um eine speziell dafür geeignete Programmiersprache handelt, können deren Sprachwerkzeuge direkt verwendet werden. Dazu zählen im einzelnen:

- Ein *Editor*, um die Prototypimplementierung der Programmbausteine zu erstellen.
- Ein *Interpreter*, um die Prototypimplementierungen ausführen zu können.
- Ein *Debugger*, um Fehler in Prototypimplementierungen zu lokalisieren.

Die erstellten Prototypimplementierungen werden mit der SEDB verwaltet.

M Evaluieren und Modifizieren des Architekturprototyps

Bevor ein Architekturprototyp geprüft werden kann, muß er ausführbar sein. Dazu werden im Normalfall die aktuellen Revisionen der einzelnen Programmbausteine in ihrer vollständigsten Implementierung zu einer Konfiguration zusammengebunden. Diese Aufgabe kann von einem *Prototypkonfigurator* geleistet werden, der aufgrund der in der SEDB gespeicherten Informationen die gewünschte Konfiguration ermittelt.

Der Architekturprototyp wird mit dem *Prototypsimulator* ausgeführt. Dieser muß sowohl Prototyp- als auch Zielsprachenimplementierungen verarbeiten können.

Spezielle Werkzeuge (*Architekturanalytoren*) können verwendet werden, um die Prüfung des Architekturprototyps gegen die Anforderungen, die an die Qualität der Architektur gestellt werden, zu unterstützen. Qualitätsmerkmale, die durch Analytoren untersucht werden können, sind beispielsweise die Anzahl der Operationen der Programmbausteine

(Schnittstellenbreite) oder der Kopplungsgrad zwischen den Bausteinen. Die Analytoren untersuchen zu diesem Zweck das in der SEDB abgelegte Architekturmodell.

Der Architektureditor und der Editor der Prototyping-Sprache werden verwendet, wenn der Architekturprototyp modifiziert werden muß.

Z Implementieren der einzelnen Programmbausteine in der Zielsprache, Testen und Integrieren in den Architekturprototyp

Aus dem Modell eines Programmbausteins kann ein Rahmen erzeugt werden, der der Syntax der Zielsprache entspricht. Dazu muß eine Abbildung zwischen Architekturbeschreibungs- und Zielsprache angegeben werden. Der Rahmen kann automatisch durch ein Werkzeug erzeugt werden (*ZS-Rahmengenerator*).

Weiterhin werden die Werkzeuge eingesetzt, die für die Zielsprache vorhanden sind (*Editor, Compiler, Debugger*). Die erstellten Zielsprachenimplementierungen werden mit der SEDB verwaltet. Um den Test der codierten Programmbausteine zu erleichtern, sind *Testwerkzeuge* hilfreich.

Die Abbildung 4.5 zeigt die Werkzeuge und den wesentlichen Datenfluß zwischen diesen und der SEDB. Zusammengehörende Werkzeuge werden in Form von Werkzeugkästen dargestellt. Diese werden anschließend beschrieben. Die drei abgebildeten Teil-Datenbanken sind nur aus optischen Gründen separat dargestellt; sie sind logisch und physisch in der SEDB enthalten.

Die drei abgebildeten Werkzeugkästen enthalten im einzelnen die folgenden Werkzeuge:

Architekturwerkzeugkasten

- Architektureditor
- Architekturanalytoren

Prototyping-Werkzeugkasten

- Editor, Interpreter und Debugger für die Prototyping-Sprache
- Prototypkonfigurator, Prototypsimulator

Zielsprachenwerkzeugkasten

- Editor, Compiler und Debugger für die Zielsprache
- Testwerkzeuge (z.B. Generator für einen Testtreiber, Soll-Ist-Vergleicher)

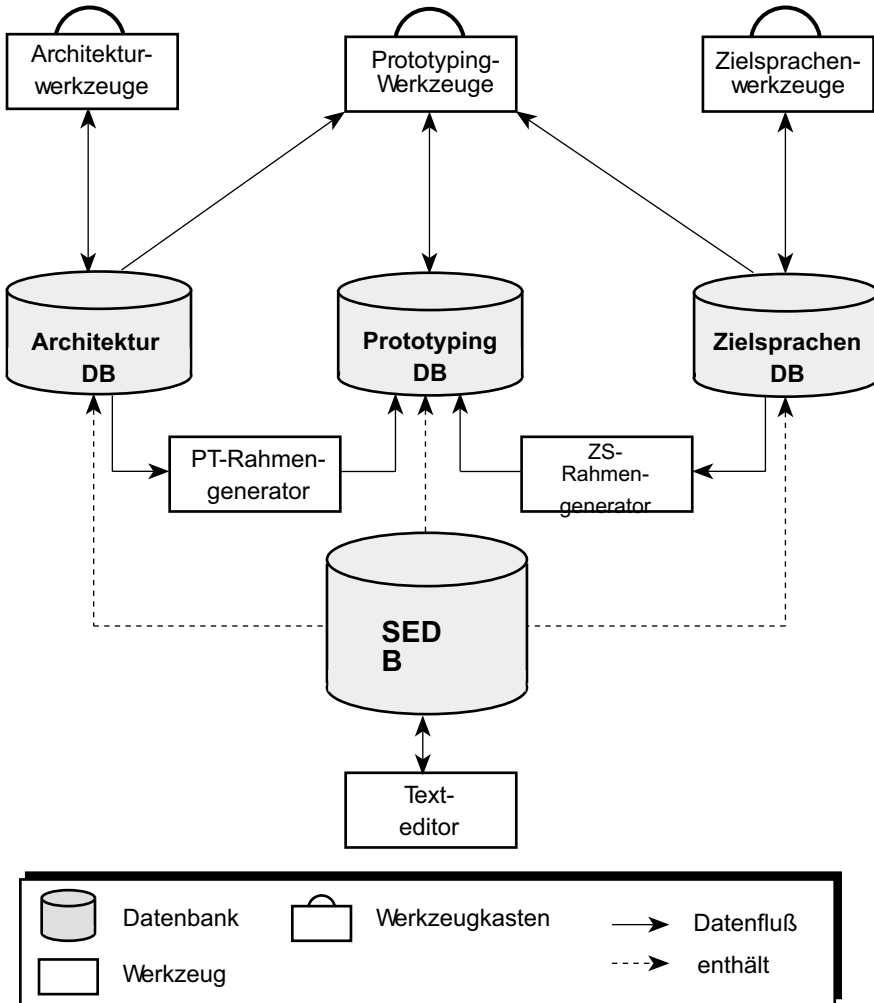


Abb. 4.5: Konzeption einer Werkstatt für PDSC

4.7 Bewertung von PDSC

PDSC zeichnet sich dadurch aus, daß die bei der Prototypentwicklung gewonnenen Informationen schrittweise in die Konstruktion des Zielprogramms einfließen. Dies wird erreicht, weil der Übergang von der Prototyp- zur Zielsprachenimplementierung schrittweise für einzelne Programmbausteine vollzogen wird. Dieser Ansatz hat folgende Vorteile:

- Wie beim Prototyping üblich, entsteht sehr schnell ein lauffähiges Programm. An diesem können die Spezifikation und der Lösungsansatz überprüft werden.
- Einmal in einer ersten Konfiguration erstellt, bleibt der Architekturprototyp ständig ausführbar. Damit entfällt die Durststrecke zwischen der Fertigstellung des Architekturprototyps und des Zielprogramms.
- Die Architektur kann schnell und leicht verändert werden, solange die Programmbausteine in einer permissiven Prototyping-Sprache implementiert sind. Entschließt man sich, auf die Zielsprache überzugehen, so gehen die Strukturinformationen nicht verloren.

Folgende Aspekte müssen bei PDSC berücksichtigt werden:

- Die einzelnen Programmbausteine können nicht in beliebiger Reihenfolge in die Zielsprache überführt werden. Dies ist ein Problem, wenn die Realisierbarkeit einzelner Programmbausteine unter gegebenen Randbedingungen, beispielsweise Effizienzanforderungen, nachgewiesen werden muß.
- PDSC kann nur zweckmäßig durchgeführt werden, wenn Entwurfsmethode, Prototyping-Sprache und Zielsprache aufeinander abgestimmt sind.

Kapitel 5

Ein Software-Schema für das Architektur-Prototyping

In Kapitel 3 wird ein dokumentenbasierter Ansatz zur Software-Modellierung beschrieben. Die Grundelemente des dazu entwickelten Software-Schemas – dem DS-Schema – sind Dokumente und Beziehungen zwischen diesen.

In diesem Kapitel wird ein Software-Schema entwickelt, das auf dem DS-Schema basiert und die im Kapitel 4 beschriebene Vorgehensweise PDSC berücksichtigt. Dieses Schema wird als Grundlage für das Software-Architektur-Prototyping verwendet.

Zuerst wird jedoch diskutiert, welche Ansätze es gibt, um Software-Schemata zu definieren, die für eine werkzeugunterstützte Modellierung geeignet sind. Anschließend wird erläutert, wie Software-Schema, Entwurfsmethode und Zielsprache aufeinander abgestimmt sein müssen. Die bei dieser Diskussion gewonnen Erkenntnisse fließen in das grundlegende Schema für das Software-Architektur-Prototyping ein.

5.1 Ansätze zur Definition von Software-Schemata

In diesem Abschnitt wird diskutiert, wie Software-Schemata definiert werden können, damit sie für den praktischen und werkzeugunterstützten Einsatz bei der Software-Entwicklung geeignet sind. Solche Schemata müssen folgende Anforderungen erfüllen:

- *Adäquate Modellierung*
Das Software-Schema muß erlauben, daß alle Attribute der Software im Modell abgebildet werden, die für die mit dem Modell untersuchte Fragestellung wichtig sind.
- *Flexible Modellierung*

Das Software-Schema darf nicht zu restriktiv, sondern muß in einem gewissen Grad flexibel sein. Dies ist erforderlich, damit auch Informationen in das Modell eingebracht werden können, die zwar für das spezielle Modell wichtig, jedoch nicht durch das Software-Schema vorgesehen sind. Ist das nicht möglich, gehen diese Informationen verloren.

Um diese beiden Anforderungen zu verdeutlichen, wird die Software-Modellierung mit dem Entwurf eines Gebäudes verglichen: Ein Architekt verwendet Schablonen, um den Plan eines Gebäudes zu entwerfen. Diese stellen diejenigen Architekturelemente zur Verfügung, die er für viele Pläne benötigt. Mit den Schablonen kann er z.B. Türen, Fenster oder Treppen zeichnen. Er kann aber zusätzlich, mithilfe von Lineal und Zirkel, Architekturelemente in den Plan einbringen, die nicht auf seinen Schablonen vorhanden sind, aber für den speziellen Grundriß benötigt werden (z.B. einen Installationsschacht).

Die Elemente einer Schablone entsprechen den Baustein- und Beziehungsarten eines Software-Schemas, die für jedes Modell verwendet werden können. Das Software-Schema muß jedoch auch „Freizeichmöglichkeiten“ im Sinn von Lineal und Zirkel vorsehen, damit es flexibel ist und in der Praxis eingesetzt werden kann.

Die beiden Anforderungen führen zu zwei verschiedenen Ansätzen, um Software-Schemata zu definieren. Der erste wird *Etikettierungsansatz*, der zweite *Spezialisierungsansatz* genannt. Diese werden nachfolgend erläutert und gegenübergestellt.

5.1.1 Der Etikettierungsansatz

Bei diesem Ansatz bietet ein Software-Schema sehr allgemeine Baustein- und Beziehungsarten an. Diese haben beschriftbare Etiketten, die verwendet werden, um einen Baustein oder eine Beziehung näher zu beschreiben. Die Etiketten werden im Schema durch Attribute der Baustein- und Beziehungsarten realisiert.

Das in Kapitel 3 definierte DS-Schema ist ein Beispiel für diesen Ansatz. Das DS-Schema definiert für die Etikettierung der Bausteinart *Dokument* das Attribut „Typ“. Dieses Attribut wird bei der Modellierung eines Dokuments mit einem Bezeichner besetzt, der die Art des Dokuments angibt. So kann dieses Attribut z.B. für das Modell der Benutzungsdokumentation mit dem Bezeichner „Handbuch“ belegt werden. Analog kann dies bei den Beziehungsarten getan werden.

5.1.2 Der Spezialisierungsansatz

Bei diesem Ansatz werden keine allgemeinen Baustein- und Beziehungsarten im Software-Schema vorgesehen, sondern es wird eine Menge spezieller Baustein- und Beziehungsarten definiert.

Dazu muß entschieden werden, welche Baustein- und Beziehungsarten für die Software-Modellierung zur Verfügung gestellt werden. Diese werden dann im Software-Schema präzise formuliert.

5.1.3 Bewertung der beiden Ansätze

Die beiden vorgestellten Ansätze werden im folgenden gegenübergestellt.

Vorteile der beiden Ansätze

Der wesentliche Vorteil des Etikettierungsansatzes besteht darin, daß beliebige Baustein- und Beziehungsarten im Modell verwendet werden können. Das Software-Schema schränkt somit die Modellierung nicht ein; es ist allgemein einsetzbar.

Beim Spezialisierungsansatz können präzise Konsistenzbedingungen formuliert werden, die für die Software-Modelle gelten. Die Konsistenzbedingungen können angegeben und mit einem Werkzeug geprüft werden, weil alle Baustein- und Beziehungsarten im Software-Schema definiert sind.

Nachteile der beiden Ansätze

Die Vorteile der Spezialisierung können als Nachteile für den Etikettierungsansatz bewertet werden. Weil hier die einzelnen Bausteine und Beziehungen durch Attributwerte charakterisiert werden, ist es nicht möglich, im Schema Konsistenzbedingungen zu formulieren, die auf die Attributwerte zugeschnitten sind. Diese Konsistenzbedingungen sind somit nur implizit vorhanden, können aber nicht durch ein Werkzeug geprüft werden.

Im Gegensatz zur Etikettierung können beim Spezialisierungsansatz nur die Baustein- und Beziehungsarten in Software-Modellen verwendet werden, die das Software-Schema explizit definiert. Andere Elementarten sind, auch wenn sie für die Modellierung benötigt werden, nicht erlaubt.

Die in Abschnitt 5.1 formulierten Anforderungen an Software-Schemata, werden erfüllt, wenn die beiden Ansätze – Etikettierung und Spezialisierung – kombiniert werden.

5.2 Programmwurfsmethode und Programmschema

Unterschiedliche Entwurfsmethoden führen nach Nagl (1990) zu unterschiedlichen Programmarchitekturarten. Betrachtet man z.B. den funktions- und den objektorientierten Programmwurf, so führt der erste zu Architekturen, deren Bausteine Funktionen und Funktionsgruppen sind. Der objektorientierte Entwurf führt, wie z.B. in Meyer (1988) beschrieben wird, zu Architekturen, die aus Klassen bestehen, die mithilfe der Vererbung strukturiert sind.

Die Zielsprache, in der eine Programmarchitektur realisiert wird, muß für die Art der Architektur geeignet sein. Da in der Regel die Zielsprache vorgegeben ist, wird die Entwurfsmethode und das Programmschema so gewählt, daß die Programmarchitekturen in der Zielsprache implementiert werden können. Bietet das Programmschema Baustein- und Beziehungsarten an, die nicht direkt in der Zielsprache formuliert werden können, so muß eine einfache, klare und praktikable Abbildung angegeben werden. Ist dies nicht möglich, so entsteht ein Bruch zwischen dem Entwurf der Architektur und deren Realisierung.

Dies verdeutlicht das folgende Beispiel: Die Beziehungsart „erbt_von“ zwischen Bausteinen kann ein Programmschema nur sinnvollerweise zur Verfügung stellen, wenn die verwendete Zielsprache objektorientiert ist. Wird jedoch in einer nicht-objektorientierten Zielsprache wie Modula-2 oder Ada implementiert, muß diese Beziehungsart mit ungeeigneten Sprachkonstrukten nachgebildet werden. Versuche dieses zu tun, führen, wie Baker (1990) zeigt, zu komplexen und unübersichtlichen Implementierungen, die nicht mehr mit dem entworfenen Modell der Programmarchitektur übereinstimmen.

Die Entwurfsmethode, das Programmschema und die verwendete Zielsprache müssen aufeinander abgestimmt sein, damit es zu keinem Bruch in der Entwicklung kommt, der letztlich die Entwicklung selbst, aber auch die Wartung der Software erschwert.

5.3 Anforderungen an ein Software-Schema für PDSC

Nachfolgend werden die Anforderungen aufgelistet, die an das PDSC-Schema gestellt werden. Diese Anforderungen ergeben sich

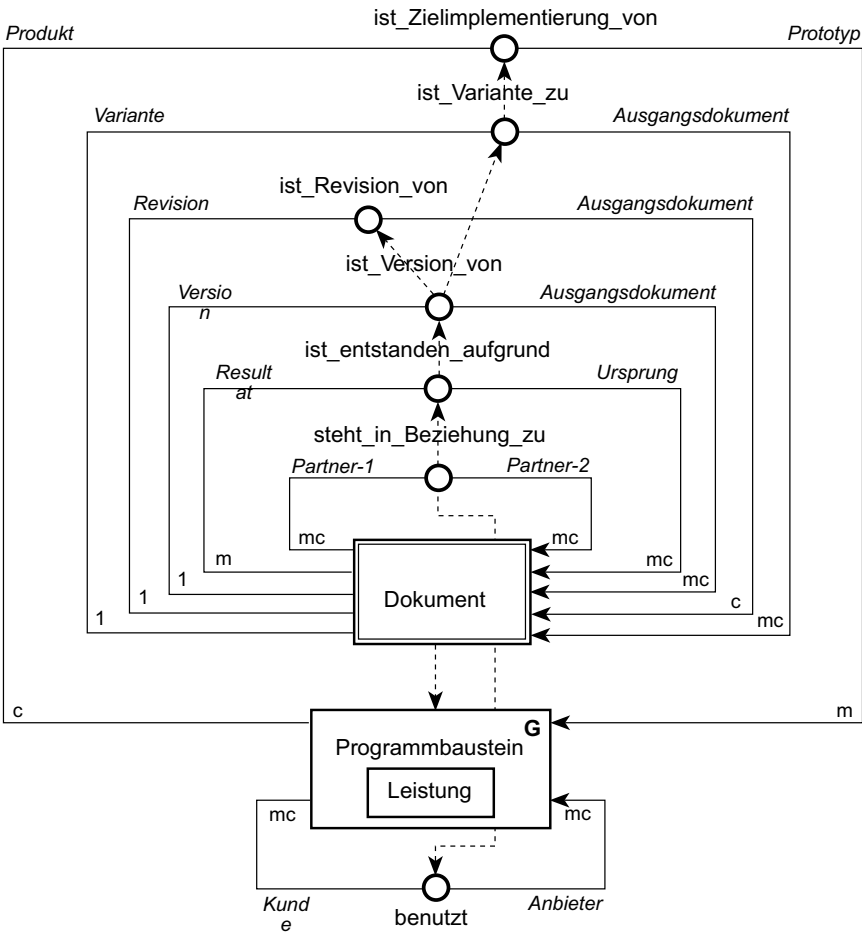
- aus den in Kapitel 4 beschriebenen Eigenschaften von PDSC,
- aus den Ergebnissen der Diskussion, wie der Programmentwurf und ein Programmschema zusammenhängen,
- aus der Bewertung der beiden Ansätze zur Schemadefinition.

Folgende Anforderungen muß das PDSC-Schema erfüllen:

- 1 Das Schema muß für die Programmarchitekturmodellierung vorsehen, daß modulare Zielsprachen verwendet werden.
- 2 Mit dem Schema muß modelliert werden können, daß Programmbausteine in Prototyp- und Zielsprachenimplementierung vorliegen.
- 3 Das Schema muß unabhängig von einer konkreten Programmentwurfsmethode und von der benutzten Zielsprache definiert sein. Diese Anforderung führt dazu, daß das Schema nur sehr allgemeine Baustein- und Beziehungsarten anbieten kann.
- 4 Das Schema muß als Basis dienen, um Software-Schemata zu definieren, die auf spezielle Programmentwurfsmethoden und Zielsprachen abgestimmt sind. Dazu muß es erweiterbar sein.
- 5 Das Schema muß vorsehen, daß vage Informationen formuliert werden können. Nach Ludewig et al. (1985b) sind vage Informationen offene, noch unbestimmte Entscheidungen. Vage Informationen müssen während der Modellierung konkretisiert und bestimmt werden können.
- 6 Das Schema muß überschaubar und im Sinn der Etikettierung flexibel sein, damit es praktikabel und anwendbar ist. Es darf nicht unnötig überladen oder komplex sein.

5.4 Ein Software-Schema für PDSC

Das PDSC-Schema ist nachfolgend abgebildet. Es erweitert das DS-Schema um Baustein- und Beziehungsarten für das Software-Architektur-Prototyping.



Beziehungsarten zwischen *Programmbaustein* und *Leistung*.

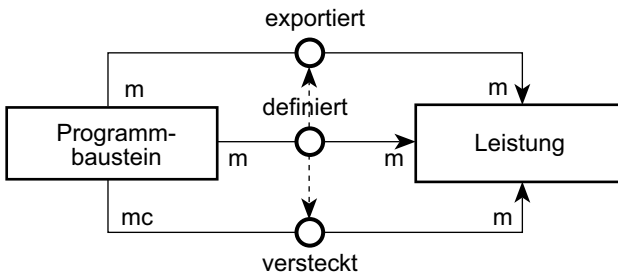


Abb. 5.1: Das PDSC-Software-Schema

5.4.1 Die Bausteinararten des Schemas

Das PDSC-Schema definiert neben der Bausteinarart *Dokument*, die bereits im DS-Schema eingeführt wurde, lediglich zwei weitere neue Bausteinararten.

Die Bausteinarart *Programmbaustein*

Sie wird in das PDSC-Schema aufgenommen, da beim Software-Architektur-Prototyping ein ausführbares Modell der Programmarchitektur erstellt wird. Die Dokumente werden mithilfe dieser Bausteinarart in zwei Gruppen unterteilt: in Dokumente, die in Maschinencode übersetzbar sind, zu einem Programm zu-sammengebunden sowie ausgeführt werden können, und in Dokumente, die diese Eigenschaft nicht haben.

Da PDSC für modulare Zielsprachen konzipiert ist, wird unter einem Programmbaustein ein Behälter verstanden, der Leistungen anbietet. Die Bausteinarart ist dementsprechend als eine Gruppierung der Bausteinarart *Leistung* definiert und ist als Generalisierung für konkrete Programmbausteinararten vorgesehen. Das Schema legt noch nicht fest, welche Programmbausteinararten, für die Modellierung zur Verfügung stehen, beispielsweise Funktionsmodul, Klasse oder Datenkapsel. Die Bausteinarart muß spezialisiert werden, wenn das PDSC-Schema an eine Programm-entwurfsmethode angepaßt werden soll, die spezielle Programmbausteinararten vorsieht.

Die Bausteinarart *Leistung*

Mit ihr werden in allgemeiner Form Leistungen modelliert, die ein Programmbaustein definiert. Diese Bausteinarart ist ebenfalls unbestimmt. Es ist noch nicht festgelegt, welche Arten von Leistungen, z.B. Prozeduren, Variablen oder Typen, vorgesehen sind. Die Bausteinarart muß spezialisiert werden, wenn konkrete Leistungen modelliert werden sollen.

5.4.2 Die Beziehungsarten des Schemas

Im folgenden werden nur die Beziehungsarten vorgestellt, die das PDSC-Schema zusätzlich zu den Beziehungsarten des DS-Schemas definiert.

ist_Zielimplementierung_von (Programmbaustein, Programmbaustein)

Damit wird modelliert, daß eine Zielsprachenimplementierung nach dem Vorbild der Prototypimplementierung entsteht. Dies ist für PDSC charakteristisch. Die Beziehungsart ist eine Spezialisierung der Beziehungsart

ist_Variante_zu (Dokument, Dokument) des DS-Schemas, weil die Zielsprachenimplementierung die vom Prototyp vorgegebene Einbindung in die Architektur nicht verändert, sondern lediglich den Baustein vollständig implementiert.

Konsistenzbedingungen

- Zu einem Programmbaustein, der ein Prototyp ist, gibt es mindestens eine Zielsprachenimplementierung. Zu einer Zielsprachenimplementierung gibt es keinen oder genau einen Baustein in Prototypimplementierung. Es existiert immer dann keine Prototypimplementierung, wenn der Programmbaustein direkt in der Zielsprache realisiert werden kann.

benutzt (Programmbaustein, Programmbaustein)

PDSC geht davon aus, daß modulare Zielsprachen verwendet werden. Modulare Sprachen sehen vor, daß Programmbausteine andere Programmbausteine benutzen. Dies wird mit dieser Beziehungsart modelliert. Sie ist eine Spezialisierung der Beziehungsart *steht_in_Beziehung_zu (Programmbaustein, Programmbaustein)*. Der erste Programmbaustein in der Beziehung ist der Kunde, der Leistungen benutzt, der zweite ist der Anbieter, der Leistungen zur Verfügung stellt.

Konsistenzbedingungen

- Programmbausteine können beliebig viele Programmbausteine benutzen und von beliebig vielen Bausteinen benutzt werden.
- Die Beziehungsart ist irreflexiv und asymmetrisch.

Die Beziehungsarten zwischen Programmbausteinen und Leistungen

Programmbausteine definieren Leistungen, die sie anderen Bausteinen anbieten oder privat sind. Dieses wird im PDSC-Schema mit drei Beziehungsarten modelliert.

Die Beziehungsart *definiert (Programmbaustein, Leistung)* ist unbestimmt. Es ist noch nicht entschieden, ob eine Leistung exportiert wird oder privat ist. Für diese Beziehungsart existieren zwei Spezialisierungen:

Mit der Beziehungsart *exportiert (Programmbaustein, Leistung)* wird ausgedrückt, daß ein Programmbaustein eine Leistung in seiner Schnittstelle zur Verfügung stellt.

Die Beziehungsart *versteckt (Programmbaustein, Leistung)* drückt aus, daß die Leistung privat ist und nicht in der Schnittstelle bereitgestellt wird.

Konsistenzbedingungen

- Ein Programmbaustein exportiert wenigstens eine Leistung, unterschiedliche Bausteine können gleiche Leistungen exportieren.
- Eine Leistung kann entweder von einem Programmbaustein exportiert oder versteckt werden.
- Ein Programmbaustein kann Leistungen verstecken, muß dies aber nicht.

5.4.3 Bewertung des Schemas

Das entwickelte Schema stellt die Baustein- und Beziehungsarten zur Verfügung, die allgemein für das Software-Architektur-Prototyping nach PDSC benötigt werden.

Das Schema muß gegen die Anforderungen geprüft werden, die es erfüllen soll. Folgendes kann festgestellt werden:

- Das Schema berücksichtigt die PDSC-spezifischen Eigenschaften: die Verwendung modularer Zielsprachen und den Übergang von einer Prototyp- zur Zielsprachenimplementierung.
- Das Schema ist unabhängig von einer Programmwurfsmethode definiert. Es unterscheidet lediglich ausführbare und nicht ausführbare Dokumente. Dies ist für das Software-Architektur-Prototyping erforderlich.
- Das Schema ist an eine spezielle Programmwurfsmethode anpaßbar. Dies geschieht dadurch, daß die Bausteinarten *Programmbaustein* und *Leistung* sowie die generellen Beziehungsarten *steht_in_Beziehung_zu* und *ist_entstanden_aufgrund* in einem erweiterten Software-Schema spezialisiert werden.
- Mit dem Schema können noch offene Entscheidungen formuliert werden. Dazu dienen die generellen Baustein- und Beziehungsarten.
- Das Schema ist übersichtlich. Es definiert nur wenige, aber allgemeine Baustein- und Beziehungsarten.

5.5 Bisher erzielte Ergebnisse

In diesem Abschnitt werden die Ergebnisse zusammengefaßt, die bisher erzielt wurden. Mit diesen Ergebnissen als Grundlage wird in den nachfolgenden Kapiteln beschrieben, wie Architekturen objektorientiert konstruierter Anwendungssoftware modelliert werden.

In Kapitel 2 wurde herausgearbeitet, daß ein Prototyp – entsprechend der Terminologie und Taxionomie, die Stachowiak in der allgemeinen Modelltheorie formuliert hat – ein Computermodell ist. Ein Prototyp und der Prozeß seiner Konstruktion haben dementsprechend auch die Merkmale, die für Modelle und das Modellerschaffen charakteristisch sind (z.B. das Verkürzungs- und das Abbildungsmerkmal). Ein bemerkenswerter Unterschied zwischen einem Prototyp und anderen Computermodellen, beispielsweise einem Programm zur Wettervorhersage oder einem Flugsimulator, besteht darin, daß das Original eines Prototyps ebenfalls ein Computerprogramm ist.

Anschließend wurde im selben Kapitel erörtert, wie Prototypen klassifiziert werden können und welche Ziele erreicht werden sollen, wenn Prototypen im Prozeß der Software-Entwicklung verwendet werden. Typisch für das Prototyping sind die Zyklen, in denen der Prototyp bewertet und modifiziert wird.

Auf der Basis dieser Vorüberlegungen wurde in Kapitel 3 die Modellierung der Software untersucht. Die Dokumente, die im Prozeß der Software-Entwicklung erstellt werden, bilden die Basis dazu. Es wurden allgemeine Beziehungsarten diskutiert, mit denen modelliert werden kann, wie die Dokumente entstehen und wie sich die einzelnen Dokumente im Sinne von Versionen weiterentwickeln. Die vorgestellten Beziehungsarten wurden in einem Software-Schema – dem DS-Schema – präzise formuliert.

Im nächsten Schritt wurde ein allgemeiner Architekturbegriff für Dokumente eingeführt und anschließend auf die Architektur der Software übertragen. Die Software-Architektur enthält aus dem gesamten Haufen der Dokumente nur diejenigen, die für den Einsatz, das Verständnis und die Wartung des Zielprogramms notwendig sind. Die Dokumente, die für den Konstruktionsprozeß der Software benötigt wurden, scheiden aus.

Diese Überlegungen führten zu einer Klassifikation von Software-Modellierungsansätzen. Diese Klassifikation definiert die Software-Architekturmodellierung als einen speziellen Ansatz der Software-Modellierung und

führt das Software-Architektur-Prototyping als Spezialisierung der Software-Architekturmodellierung ein. Charakteristisch für das Software-Architektur-Prototyping ist, daß dabei ein Modell der Programmarchitektur entsteht, das ausgeführt werden kann.

Im Kapitel 4 wurde eine Vorgehensweise für das Software-Architektur-Prototyping vorgestellt, die PDSC genannt wurde. Sie sieht vor, daß der konstruierte Architekturprototyp schrittweise in das Zielprogramm überführt wird. Es wurde diskutiert, warum dies sinnvoll ist und welche Vorteile diese Vorgehensweise hat. Weiterhin wurde beschrieben, welche Aktivitäten bei PDSC durchgeführt werden müssen, wie der Übergang im einzelnen vonstatten geht, welche Sprachen dazu notwendig sind und welche Werkzeuge diese Vorgehensweise operational unterstützen.

In diesem Kapitel wurde das DS-Schema erweitert, damit es an die Erfordernisse der Vorgehensweise PDSC angepaßt ist. Das resultierende Software-Schema ist das PDSC-Schema. Mit diesem Schema kann die Software gemäß PDSC modelliert werden; es ist jedoch noch nicht an eine spezielle Entwurfsmethode angepaßt.

Kapitel 6

Prototyping objektorientiert entworfener Architekturen

In diesem Kapitel wird das Instrumentarium geschaffen, um die in den vorangegangenen Kapiteln formulierten Ideen für das Architektur-Prototyping auf der Basis einer konkreten Software-Entwurfsmethode umzusetzen. Dazu wird der objektorientierte Ansatz zur Software-Entwicklung gewählt.

Zuerst wird das vorgestellte PDSC-Schema so erweitert, daß mit ihm objektorientiert konstruierte Anwendungssoftware modelliert werden kann. Anschließend werden die Sprachen eingeführt, mit denen entsprechende Architekturmodelle formuliert und die konstruierten Architekturprototypen realisiert werden.

6.1 Architektur-Prototyping im Überblick

Bevor das Software-Schema, die Sprachen und Werkzeuge für das Architektur-Prototyping objektorientierter Anwendungen eingeführt werden, wird in diesem Abschnitt ein Überblick gegeben, der alle „Komponenten“ in Beziehung setzt.

6.1.1 Konzepte, Methoden, Sprachen und Werkzeuge

In Abschnitt 2.1.1 werden Konzepte, Sprachen, Methoden und Werkzeuge als Teile eines Systemdreiecks betrachtet. Für das Architektur-Prototyping objektorientierter Anwendungen kann folgendes konkrete Systemdreieck angegeben werden.

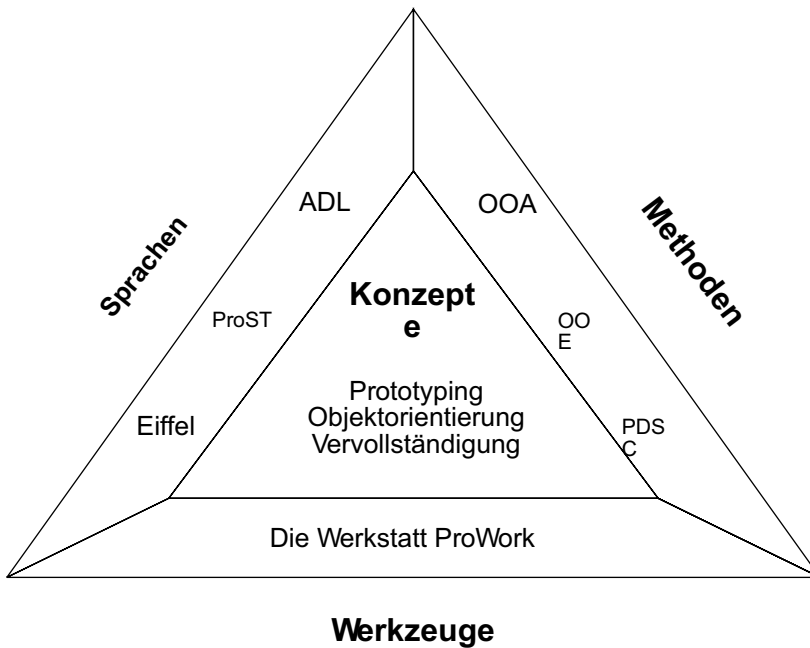


Abb. 6.1: Systemdreieck für das Architektur-Prototyping objektorientierter Anwendungen

Konzepte

Die zentralen Konzepte sind Prototyping, Vervollständigung und Objektorientierung¹. Daneben fließen weitere Konzepte in diesen Ansatz ein, beispielsweise das Konzept, Software mit Entitäten und Beziehungen zu modellieren.

Methoden

Neben der Vorgehensweise PDSC, die in Kapitel 4 vorgestellt wird, sind Methoden notwendig, um die Aufgabenstellung zu analysieren und das Programm zu entwerfen.

¹ Prototyping wird eingehend in Abschnitt 2.3 beschrieben. Die Vervollständigung wird in Abschnitt 4.2.1 erläutert. Die Objektorientierung wird in Abschnitt 6.2.3 vorgestellt.

Da Prototypen objektorientiert konstruierter Anwendungen erstellt werden sollen, werden Methoden zur objektorientierten Analyse (OOA) und zum objektorientierten Entwurf (OOE) benötigt. Wenn auch bisher *die* OOA- und *die* OOE-Methode fehlen, so existieren doch verschiedene Ansätze, die ein methodisches Vorgehen definieren. Bekannt sind die Analysemethoden von Shlare und Mellor (1988) und von Coad und Yourdon (1991). In Champeaux (1992) werden die bekannten Methoden miteinander verglichen. Auf dem Gebiet des objektorientierten Entwurfs sind die Methoden von Booch (1991), Budde et al. (1992a) und Rumbaugh (1991) zu nennen.

In dieser Arbeit werden spezielle OOA- und OOE-Methoden nicht näher erläutert; es wird auf die oben genannten Literaturangaben verwiesen.

Ein Vorteil der objektorientierten Software-Konstruktion besteht darin, daß kein Bruch zwischen Analyse, Entwurf und Implementierung entsteht, weil bei allen Aktivitäten mit denselben Abstraktionsmitteln gearbeitet wird.

Bei der objektorientierten Analyse werden die wesentlichen Gegenstände des Anwendungsgebiets, die sowohl konkret als auch abstrakt sein können, identifiziert und in Begriffen zusammengefaßt. Diese Begriffe werden definiert und strukturiert. Dazu werden auf der einen Seite Merkmale gesucht, die einen Begriff charakterisieren. Auf der anderen Seite werden Umgangsformen gesucht, die beschreiben, wie die Gegenstände, die letztlich hinter den Begriffen stehen, typischerweise gebraucht werden. Auf dieser Basis wird beim objektorientierten Entwerfen eine software-technische Lösung entwickelt. Dabei wird von den Begriffen ausgegangen, um die Klassen der Programmarchitektur zu definieren. Bei dieser Vorgehensweise werden somit aus den Begriffen der Anwendung die software-technischen Komponenten (Klassen) der Programmarchitektur.

Sprachen und Werkzeuge

Es werden die folgenden Sprachen für das Architektur-Prototyping objektorientierter Anwendungssoftware verwendet: die Software-Modellierungssprache ADL, die Prototyping-Sprache ProST und die Zielsprache Eiffel¹. Die zur Verfügung gestellten Werkzeuge sind in der Werkstatt ProWork zusammengefaßt, die in Kapitel 7 beschrieben wird.

¹ ADL wird in Abschnitt 6.6 und ProST in Abschnitt 6.7 vorgestellt.

6.1.2 Tätigkeiten

Die Tätigkeiten, die bei PDSC durchzuführen sind, können für das Architektur-Prototyping objektorientierter Anwendungssoftware konkretisiert werden. Dazu werden die gewählten Methoden und Sprachen in das Tätigkeitenmodell aus Abschnitt 4.2.3 integriert. Die folgende Abbildung zeigt das Resultat.

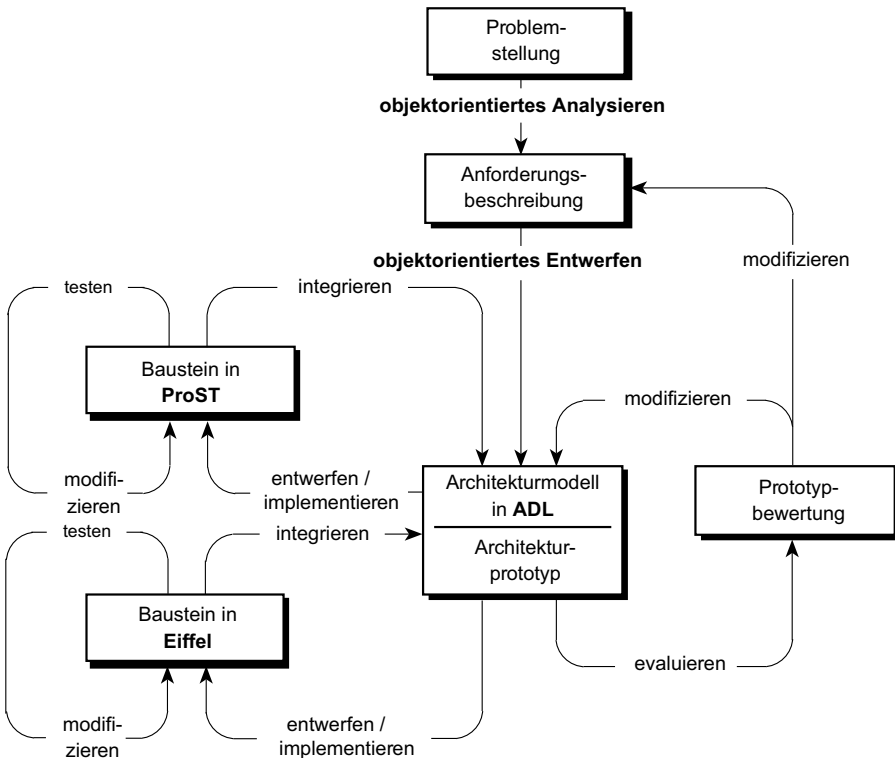


Abb. 6.2: Tätigkeiten beim Prototyping objektorientierter Anwendungssoftware

6.2 Ein Software-Schema für objektorientierte Programmarchitekturen

Das PDSC-Schema kann, so wie es definiert ist, nicht eingesetzt werden, um konkrete Software-Architekturen zu modellieren. Dazu ist es zu unbestimmt. Nachfolgend wird am Beispiel der objektorientierten Programm-entwicklung gezeigt, wie das PDSC-Schema zu einem praktikablen

Software-Schema erweitert werden kann. Das resultierende Schema wird *OOS-Schema* (**O**bjektorientiertes **S**oftware-Schema) genannt.

6.2.1 Vorbemerkungen

Während DS- und PDSC-Schema schrittweise und systematisch auf der Grundlage der dokumentenbasierten Software-Modellierung und der Vorgehensweise PDSC entwickelt wurden, wird das im folgenden vorgestellte OOS-Schema so definiert, daß es in den offenen Rahmen, den das PDSC-Schema anbietet, eingepaßt werden kann.

Das OOS-Schema wird dazu aus bekannten objektorientierten Modellierungselementen zusammengesetzt. Es basiert weitgehend auf den Konzepten der Programmiersprache Eiffel, die in Meyer (1992) beschrieben wird. Dies hat folgende Gründe:

- Eiffel bietet Konzepte, die nach Budde et. al (1992b) bereits beim Entwurf verwendet werden können.
- Die mit Eiffel modellierbaren Architekturen können auf andere objektorientierte Sprachen abgebildet werden, die in der Praxis eingesetzt werden (z.B. C++).

Das entwickelte OOS-Schema ist nicht das einzig denkbare, um objektorientierte Programmarchitekturen zu modellieren. Es ist auf der einen Seite so definiert, daß die charakteristischen Baustein- und Beziehungsarten objektorientierter Architekturen modelliert werden können. Auf der anderen Seite wird aber auf Modellierungselemente verzichtet, die in einem anderen Kontext durchaus sinnvoll sind. Dazu zählt beispielsweise die Mehrfachvererbung.

Dies wurde aus zwei Gründen getan: Erstens soll das Schema übersichtlich bleiben. Zweitens soll es als Kern verwendet werden, um im Rahmen dieser Arbeit beispielhaft Sprachen und Werkzeuge für das Software-Architektur-Prototyping zu entwickeln.

6.2.2 Warum eine objektorientierte Erweiterung?

In den letzten Jahren wurde die objektorientierte Software-Entwicklung besonders intensiv untersucht und weiterentwickelt. Wesentliche Grundlagen für den objektorientierten Ansatz zur Software-Entwicklung bilden die Arbeiten über *Information Hiding* von David Parnas (1972) und über die *Abstrakten Datentypen* von Barbara Liskov (1974). Der Begriff *objektorientiert* wird in dieser Arbeit gemäß der Definition von Wegner (1987) verwendet.

Der objektorientierte Ansatz zur Software-Entwicklung wurde aus folgenden Gründen als Basis für das Software-Architektur-Prototyping gewählt:

- Er ist nach Meinung vieler Autoren, z.B. Booch (1991) oder Coad (1992), geeignet, um Programmarchitekturen zu erstellen, die wartbar, portabel und erweiterbar sind. Diese Qualitätsanforderungen existieren nach Boehm (1976) für jede Software.
- Objektorientierte Sprachen, besonders C++ und Smalltalk-80, werden bereits in der industriellen Software-Entwicklung eingesetzt (siehe dazu Kanath, 1992), oder es wird geplant, objektorientierte Sprachen in absehbarer Zeit als unternehmensweite Entwicklungssprachen einzusetzen.¹
- Er kann die Produktivität des Entwicklungsprozesses und die Qualität der Produkte verbessern, wenn wiederverwendbare Programmbausteine konstruiert und in anderen Projekten verwendet werden. Dies haben Lewis et al. (1992) in einer empirischen Studie gezeigt. Allerdings wird die Wiederverwendung nicht automatisch dadurch verbessert, daß objektorientiert konstruiert wird. So gibt es nach Taenzer (1989) auch bei diesem Ansatz Probleme, wenn wiederverwendbare Programmbausteine konstruiert werden sollen.
- Nicht zuletzt wurde dieser Ansatz gewählt, weil objektorientierte Programmiersprachen zur Verfügung standen und weil eigene positive Erfahrungen mit der objektorientierten Software-Entwicklung gemacht wurden.

6.2.3 Ein Schema für objektorientierte Programmarchitekturen

Objektorientierte Programmarchitekturen bestehen aus einer Ansammlung von *Klassen*. Eine Klasse ist eine Schablone, nach der Objekte erzeugt werden können. Sie definiert deren Struktur und Verhalten. Während Klassen die Bausteine der Architektur sind, sind die Objekte die Elemente, die zur Laufzeit des Programms existieren.

¹ Auf einem Symposium über objektorientierte Software-Entwicklung, das von der IBM Deutschland veranstaltet wurde und vom 6. bis 8. November 1990 in Herrenberg stattgefunden hat, wurde C++ als die zukünftige strategische Entwicklungssprache der IBM genannt. Smalltalk-80 soll für das Prototyping verwendet werden.

Klassen *definieren Eigenschaften*, die *exportiert* werden können. Die Eigenschaften werden in *Merkmale* und *Routinen* unterteilt. Merkmale charakterisieren die Objekte einer Klasse; Routinen realisieren deren Verhalten. Eine spezielle Form der Routine ist die *Funktion*, da sie immer einen Ergebniswert zurückliefert.

Klassen können auf zwei unterschiedliche Arten miteinander in Beziehung gesetzt werden:

- Eine Klasse kann andere Klassen *benutzen*, um ihre eigenen Routinen zu implementieren.
- Eine Klasse kann von anderen Klassen *erben*.

Die *Vererbung* ist die für den objektorientierten Ansatz charakteristische Beziehung, um Klassen zu strukturieren. Eine Klasse erbt alle Eigenschaften der vererbenden Klasse. Die Vererbung führt zu Klassenhierarchien, die im Fall der Einfachvererbung als Bäume und im Fall der Mehrfachvererbung als Netze dargestellt werden können.

Geerbte Eigenschaften können *redefiniert* werden. Mithilfe der Vererbung ist es möglich, gemeinsame Eigenschaften verschiedener Klassen in einer eigenen Klasse zu definieren, von der diese Klassen dann erben. Durch diesen Prozeß entstehen Klassen, die die gemeinsamen Eigenschaften nur noch spezifizieren, aber nicht mehr realisieren können. Diese werden *Virtuelle Klassen*¹ genannt. Routinen, die nicht implementiert werden können, werden entsprechend *Virtuelle Routinen* genannt.

Eine weitere spezielle Form einer *Klasse* ist die *Generische Klasse*. Eine generische Klasse ist ein flexibler Programmbaustein, der mithilfe von Parametern auf einen speziellen Zweck angepaßt werden kann. Dazu besitzen generische Klassen sogenannte formale generische Typparameter. Beispiele für generische Klassen sind die in vielen Klassenbibliotheken angebotenen Behälter- oder Container-Klassen. Dazu gehören beispielsweise die Klassen „Liste“ oder „Menge“. Der Inhalt eines Behälters, z.B. einer Liste, kann über einen generischen Typparameter bestimmt werden.

Ist eine Klasse sowohl generisch als auch virtuell, wird sie folglich als *Virtuelle & Generische Klasse* bezeichnet.

¹ Virtuelle Klassen werden auch *aufgeschobene* oder *abstrakte* Klassen genannt.

6.3 Das OOS-Schema

Abbildung 6.3 zeigt, wie die in dieser Arbeit vorgestellten Software-Schemata zusammenhängen.

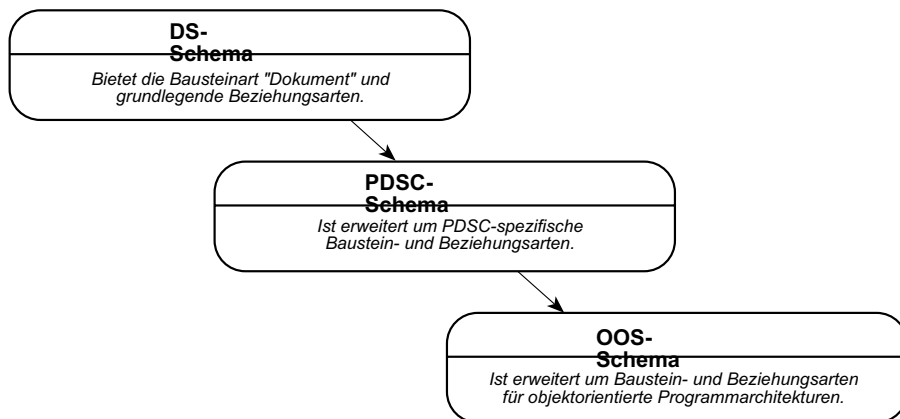


Abb. 6.3: Entwicklung der Software-Schemata

Das OOS-Schema entsteht dadurch, daß das im vorigen Abschnitt beschriebene Schema objektorientierter Programmarchitekturen in das vorhandene PDSC-Schema integriert wird. Als Ansatzpunkte dienen die Bausteinararten *Programmbaustein* und *Leistung* sowie die Beziehungsarten *ist_entstanden_aufgrund* und *steht_in_Beziehung_zu*. Die neuen Baustein- und Beziehungsarten des OOS-Schemas werden nachfolgend im einzelnen erläutert.

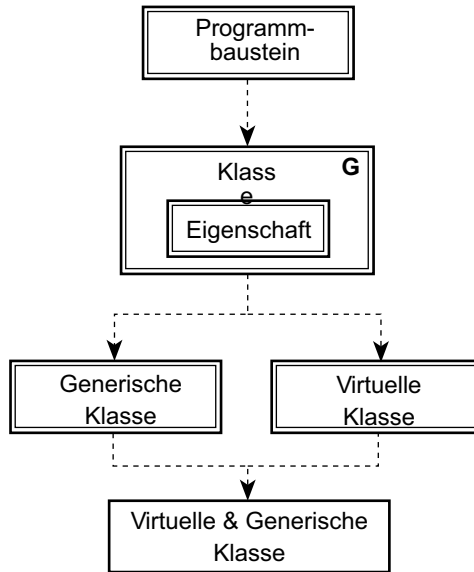
6.3.1 Die Bausteinararten des OOS-Schemas

Die Bausteinarart *Klasse* und ihre Spezialisierungen

Die Bausteinarart *Klasse* ist nach der in Abschnitt 5.5.1 angegebenen Bedeutung ein spezieller Programmbaustein.¹ Eine Klasse ist deshalb im

¹ Die Baustein-Metapher ist jedoch in diesem Fall nicht besonders angemessen, da eine Klasse im Sinn eines abstrakten Datentyps auch als Bausteinfabrik angesehen werden kann, die – vergleichbar mit einer Ziegelsteinfabrik – gleichartige Bausteine herstellen kann.

Schema als Spezialisierung der Bausteinart *Programmbaustein* des PDSC-Schemas definiert und ist eine Gruppierung von Eigenschaften.



Die Bausteinarten *Virtuelle Klasse* und *Generische Klasse* sind Spezialisierungen davon. Eine virtuelle Klasse ist dadurch charakterisiert, daß sie wenigstens eine virtuelle Routine besitzt. Eine generische Klasse hat formale generische Typparameter.

Die Bausteinart *Virtuelle & Generische Klasse* vereinigt die Eigenschaften der *Virtuellen Klasse* und der *Generischen Klasse*.

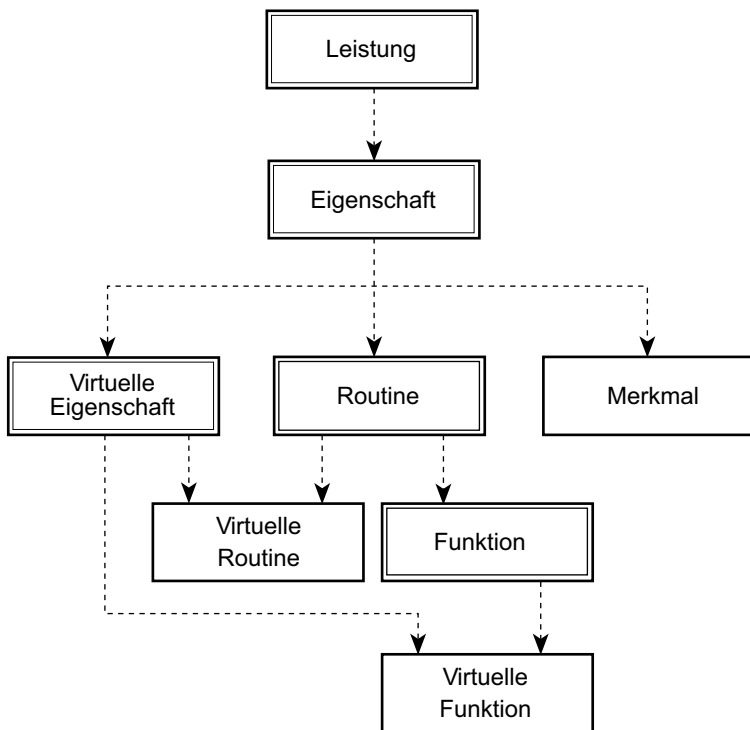
Die Bausteinart *Eigenschaft* und ihre Spezialisierungen

Klassen definieren Eigenschaften. Diese werden in allgemeiner Art mit der Bausteinart *Eigenschaft* modelliert. Sie ist eine Spezialisierung der Bausteinart *Leistung* des PDSC-Schemas. Sie definiert das Attribut „Bezeichner“, das für alle Arten von Eigenschaften benötigt wird. Die Bausteinart ist jedoch noch unbestimmt und wird in die Bausteinarten *Virtuelle Eigenschaft*, *Routine* und *Merkmal* spezialisiert.

Mit der Bausteinart *Virtuelle Eigenschaft* wird modelliert, daß eine Eigenschaft nur spezifiziert, jedoch nicht realisiert werden kann.

Ein Merkmal ist eine Eigenschaft, die zusätzlich zum Attribut „Bezeichner“ ein Attribut besitzt, das den Typ des Merkmals angibt.

Die Bausteinart *Routine* dient dazu, operationale Eigenschaften zu modellieren. Dazu definiert sie ein Attribut „Parameterliste“, das die Eingabeparameter aufnimmt. Sie wird in die Bausteinarten *Funktion* und *Virtuelle Routine* weiterverfeinert.



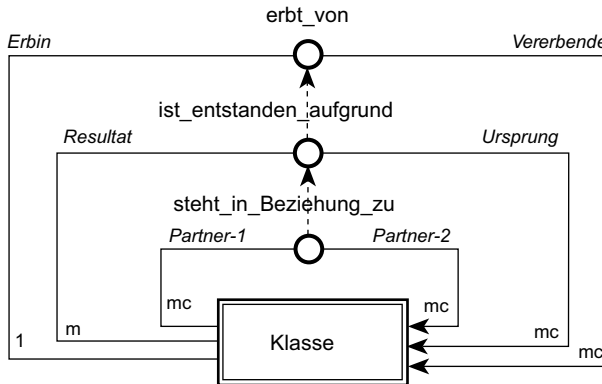
Die Bausteinart *Funktion* definiert das Attribut „Ergebnistyp“.

Die Bausteinart *Virtuelle Routine* ist sowohl eine *Routine* als auch eine *Virtuelle Eigenschaft*. Sie dient dazu, Routinen zu modellieren, die nicht realisiert werden können.

Die Bausteinart *Virtuelle Funktion* ist als Spezialisierung der Bausteinarten *Funktion* und *Virtuelle Eigenschaft* definiert. Damit modelliert sie Funktionen, die nicht implementiert werden können.

6.3.2 Die Beziehungsarten des OOS-Schemas

Die Beziehungsart *erbt_von* (Klasse, Klasse)



Das OOS-Schema muß natürlich diese Beziehungsart enthalten. Sie ist charakteristisch für alle Schemata objektorientierter Architekturen. Sie ist eine Spezialisierung der Beziehungsart *ist_entstanden_aufgrund* des PDSC-Schemas. Die erste Klasse in der Beziehung hat die Rolle der Erbin, die zweite die der Vererbenden.

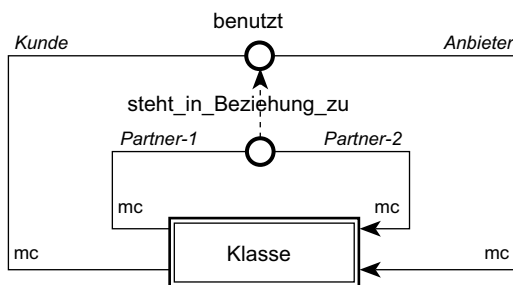
Konsistenzbedingungen

- Eine Klasse erbt genau von einer anderen Klasse, kann aber selbst Vererbende für beliebig viele andere Klassen sein. Dies entspricht der Einfachvererbung.
- Die Beziehungsart ist irreflexiv und azyklisch.
- Eine *erbt_von*-Beziehung darf nur dann zwischen Klassen eingerichtet werden, wenn eine der folgenden Bedingungen gilt:
 - Erbin und Vererbende sind vom gleichen Typ, z.B. generische Klassen.
 - Die Vererbende ist eine virtuelle Klasse.
 - Die Erbin ist eine generische Klasse und die Vererbende ist eine Klasse.
 - Die Erbin ist eine generische Klasse und die Vererbende ist eine virtuelle und generische Klasse.

Dadurch sind die folgenden Kombinationen für die Vererbung zwischen Klassen definiert; alle anderen sind ausgeschlossen:

- erbt_von (Klasse, Klasse)*
- erbt_von (Klasse, Virtuelle Klasse)*
- erbt_von (Generische Klasse, Klasse)*
- erbt_von (Generische Klasse, Virtuelle Klasse)*
- erbt_von (Generische Klasse, Generische Klasse)*
- erbt_von (Generische Klasse, Virtuelle & Generische Klasse)*
- erbt_von (Virtuelle Klasse, Virtuelle Klasse)*
- erbt_von (Virtuelle & Generische Klasse, Virtuelle Klasse)*
- erbt_von (Virtuelle & Generische Klasse, Virtuelle & Generische Klasse)*

Die Beziehungsart *benutzt* (Klasse, Klasse)

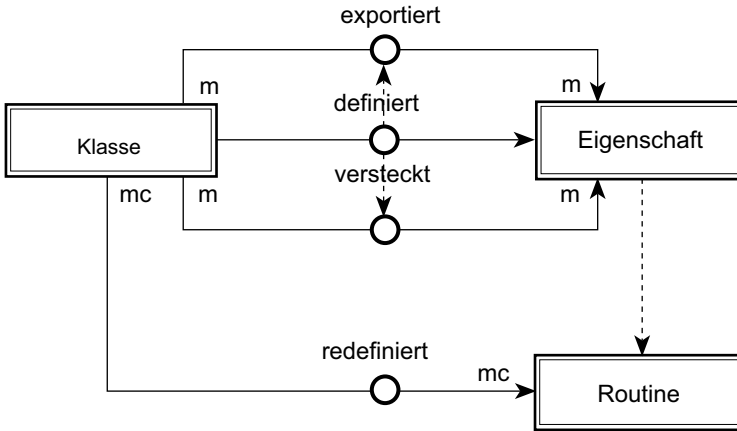


Die *benutzt*-Beziehungsart wird bereits im allgemeinen PDSC-Schema definiert. Die Konsistenzbedingungen, die dort für die Beziehungsart definiert sind, müssen im OOS-Schema angepaßt werden, weil die beteiligten Programmbausteine Klassen sind.

Konsistenzbedingungen

- Klassen können beliebig viele Klassen benutzen und von beliebig vielen Klassen benutzt werden.
- Die Beziehungsart ist irreflexiv und asymmetrisch.
- Virtuelle Klassen können zwar als Kunde, nicht aber als Anbieter in einer *benutzt*-Beziehung auftreten.
- Eine vererbende Klasse darf ihre Erben nicht benutzen.

Die Beziehungsarten zwischen Klassen und Eigenschaften



Die im PDSC-Schema definierten Beziehungsarten *definiert* (*Programmbaustein, Leistung*), *exportiert* (*Programmbaustein, Leistung*) und *versteckt* (*Programmbaustein, Leistung*) werden im OOS-Schema dadurch konkretisiert, daß Klassen konkrete Programmbausteine und Eigenschaften konkrete Leistungen sind.

Die zusätzliche Beziehungsart *redefiniert* (*Klasse, Routine*) drückt aus, daß eine Klasse eine geerbte Routine mit einer neuen Implementierung überschreibt. Die Sichtbarkeit der Routine wird dadurch nicht verändert. Wenn beispielsweise die geerbte Routine exportiert wird, wird auch die redefinierte Routine exportiert. Geerbte Merkmale können nicht redefiniert werden.

Die Konsistenzbedingungen, die für diese Beziehungsarten im PDSC-Schema definiert sind, müssen erweitert werden, da nun konkrete Programmbausteine und Leistungen in Beziehung gesetzt werden.

Konsistenzbedingungen

- Eine Klasse exportiert wenigstens eine Eigenschaft, unterschiedliche Klassen können gleiche Eigenschaften exportieren.
- Eine Eigenschaft kann entweder von einer Klasse exportiert oder versteckt werden.
- Eine Klasse kann Eigenschaften verstecken, muß dies aber nicht.

- Eine Klasse darf eine Eigenschaft nur dann definieren, wenn diese nicht bereits in der durch die *erbt_von*-Beziehung gebildeten Hierarchie über dieser Klasse definiert ist.
- Eine Klasse darf eine Routine nur dann redefinieren, wenn die Klasse diese erbt, d.h. wenn diese bereits in der Hierarchie über dieser Klasse definiert oder redefiniert ist.
- Eine Klasse kann geerbte Routinen redefinieren, muß dies aber nicht tun.
- Eine Routine einer Klasse kann in beliebig vielen erbenden Klassen redefiniert werden.
- Eine virtuelle Klasse muß mindestens eine virtuelle Routine definieren.

6.3.3 Bewertung des Schemas

Das OOS-Schema erweitert das PDSC-Schema um Elemente, die für die Modellierung objektorientierter Programmarchitekturen zur Verfügung gestellt werden. Das OOS-Schema kann bezüglich der folgenden Aspekte bewertet werden: Ist es geeignet, um Programmarchitekturen zu modellieren, die allgemeine Qualitätsmerkmale wie *Wartbarkeit* und *Wiederverwendung* besitzen? Welche Möglichkeiten stellt es zur Verfügung, die den Prozeß der Modellierung unterstützen?

Bewertung der modellierbaren Architekturen

- *Die Qualitätsmerkmale Information Hiding, Zusammenhalt und Kopplung*

Ein hoher innerer Zusammenhalt sowie eine lose Kopplung zwischen den Programmbausteinen ist nach Constantine und Yourdon (1979) ein Ziel, das beim Entwurf angestrebt werden muß.

Eine Klasse ist im Sinn eines abstrakten Datentyps ein in sich abgeschlossener Bestandteil einer Programmarchitektur. Eine Klasse hat einen hohen inneren Zusammenhalt, da sie nur die Eigenschaften definiert, die ihre Objekte charakterisieren. Objekte kommunizieren ausschließlich über Routinen, wodurch eine lose Kopplung erzielt wird.

- *Die Qualitätsmerkmale Wiederverwendung und Lokalität*

Durch systematisches Wiederverwenden können die Kosten für die Entwicklung gesenkt werden, da nach Tracz (1987) die Produktivität, aber auch die Qualität der Software verbessert wird. Dadurch, daß existierende Programmbausteine wiederverwendet werden, wird verhindert, daß bereits bekannte Bausteine immer wieder neu entwickelt und die dabei gemachten Fehler immer wieder begangen werden.

Die Vererbung ist eine geeignete Möglichkeit, um den Grad der Wiederverwendung zu erhöhen, da Eigenschaften geerbt werden und nicht noch einmal realisiert werden müssen. Durch virtuelle Klassen werden gemeinsame Eigenschaften von Klassen in einer einzigen Klasse definiert und anschließend vererbt. Dadurch werden die Auswirkungen von Änderungen und Erweiterungen lokal begrenzt. Mit generischen Klassen werden flexible Bausteine modelliert, die über Parameter angepaßt werden können und dadurch wiederverwendbar sind.

- *Das Qualitätsmerkmal Wartbarkeit*

Die Wartbarkeit einer Software ist ein Qualitätsmerkmal, das besonders wichtig ist, da nach Sommerville (1992) die Kosten für die Wartung großer Software-Systeme etwa zwei- bis viermal so hoch sind wie die Kosten für die Entwicklung.

Hoher Zusammenhalt und lose Kopplung sowie Lokalität wirken sich positiv auf die Wartbarkeit aus. Klassen können mit geringem Aufwand modifiziert, portiert oder erweitert werden, weil zusammengehörende Teile in einem einzigen Programmbaustein enthalten sind und nicht über viele verschiedene Programmbausteine verteilt sind.

Bewertung der Möglichkeiten, die den Modellierungsprozeß unterstützen

- Das Schema erlaubt, vage Informationen zu formulieren. Es ist möglich, noch offene Entscheidungen auszudrücken, die später im Modellierungsprozeß konkretisiert werden können. Dazu enthält das Schema generalisierte Baustein- und Beziehungsarten. So kann z.B. eine Klasse zu einer generischen Klasse spezialisiert werden, wenn erkannt wird, daß die Klasse flexibel entworfen werden kann.

6.4 Sprachen für das Architektur-Prototyping

Bei PDSC werden Sprachen für unterschiedliche Zwecke benötigt. Die in Abschnitt 4.4.1 formulierten Anforderungen müssen berücksichtigt werden, wenn PDSC-Sprachen ausgewählt werden. Dies gilt natürlich auch, wenn Architekturprototypen für objektorientierte Anwendungssoftware erstellt werden sollen. Nachfolgend wird erläutert, welche Sprachen dazu benutzt werden und warum diese für den entsprechenden Zweck geeignet sind.

6.4.1 Die Software-Modellierungssprache

Die Software-Modellierungssprache muß erlauben, daß Software-Modelle formuliert werden können, die dem OOS-Schema entsprechen. Da eine solche Sprache nicht existiert, wird in Abschnitt 6.6 eine Software-Modellierungssprache für das OOS-Schema vorgestellt. Diese Sprache wird *ADL* (**A**rchitecture **D**escription **L**anguage) genannt.

6.4.2 Die Prototyping-Sprache

Die Prototypimplementierungen der Programmbausteine müssen in einer objektorientierten Sprache formuliert werden, die speziell für das Prototyping geeignet ist.

Smalltalk-80 ist eine objektorientierte Sprache, die nach Sandberg (1987), Rotz (1987) und nach eigenen Erfahrungen viele Anforderungen erfüllt, die an eine Prototyping-Sprache gestellt werden:

- Smalltalk-80 ist inkrementell übersetzbar. Der erzeugte Code wird jedoch interpretiert. Diese Strategie hat den Vorteil, daß auch unvollständige Programme ausgeführt werden können.
- Smalltalk-80 hat kein Typsystem und damit auch keine strenge Typprüfung.
- Mit „ObjectWorks“ existiert eine Programmierumgebung für Smalltalk-80, die eine umfangreiche Bibliothek wiederverwendbarer Klassen enthält und Werkzeuge anbietet, mit denen sehr schnell Prototypen erstellt werden können.

Weil das OOS-Schema Eigenschaften- und Klassenarten vorsieht, die in Smalltalk-80 nicht explizit formuliert werden können (z.B. die virtuellen Klassen), kann Smalltalk-80 nicht unverändert verwendet werden. Smalltalk-80 wird deshalb erweitert. Die resultierende Sprache wird *ProST* genannt und in Abschnitt 6.7 vorgestellt.

6.4.3 Die Zielsprache

Als Zielsprache wird die objektorientierte Programmiersprache *Eiffel*, die in Meyer (1992) vorgestellt wird, eingesetzt. Dies hat folgende Gründe:

- Das OOS-Schema definiert Programmbausteinararten, die direkt in der Sprache Eiffel implementiert werden können. Eine einfache Abbildung

zwischen Software-Schema und Zielsprache ist nach Abschnitt 5.2 eine Voraussetzung für das Architektur-Prototyping.

- Eiffel ist eine Sprache, die nicht nur objektorientiert ist, sondern auch Programmierkonzepte zur Verfügung stellt, die sie aus der Sicht der Software-Technik als Zielsprache qualifizieren. Dazu zählen die Ausnahmebehandlung, das Konzept der Zusicherungen, die strenge Typprüfung und die separate Übersetzbarkeit.
- Ein Architekturprototyp muß ausführbar sein, auch wenn er aus Prototyp- und Zielsprachenimplementierungen besteht. Dies läßt sich für ProST und Eiffel realisieren. In Abschnitt 7.3 wird beschrieben, wie Programmbausteine, die in beiden Sprachen formuliert sind, miteinander verbunden werden können.

6.5 Konzeption der Sprachen ADL und ProST

Die Sprache ADL muß neu, die Sprache ProST auf der Basis von Smalltalk-80 entwickelt werden. In diesem Abschnitt wird diskutiert, welche Beziehungen zwischen diesen beiden Sprachen und der Zielsprache Eiffel bestehen und welchen Einfluß Eiffel und Smalltalk-80 auf den Entwurf der Sprachen ADL und ProST haben.

6.5.1 Beziehungen zwischen den Sprachen

Jede der vorgesehenen Sprachen für das Prototyping objektorientierter Anwendungssoftware wird für einen speziellen Zweck benötigt:

- Die Architekturbeschreibungssprache ADL, um das Software-Architekturmodell zu formulieren.
- Die Prototyping-Sprache ProST, um die Prototypimplementierungen der Programmbausteine zu erstellen, die dann den Architekturprototyp bilden.
- Die Zielsprache Eiffel, um die Programmbausteine für das Zielprogramm zu implementieren.

Die Methode PDSC legt fest, daß das Modell der Programmarchitektur schrittweise zum Zielprogramm überführt wird. Dazu werden die Programmbausteine zuerst in der Architekturbeschreibungssprache formuliert, anschließend prototypisch realisiert und evaluiert. Wenn ein Programmbaustein stabil ist, wird er in der Zielsprache vollständig implementiert.

Diese Vorgehensweise führt auf der Sprachseite dazu, daß ein Programmbaustein in der Regel in drei Formulierungen vorliegt: in der ADL-, der ProST- und der Eiffel-Formulierung.

Folgende Transformationen zwischen den Formulierungen sind sinnvoll und erforderlich:

Ein in ADL formulierter Programmbaustein muß

- in eine ProST-Formulierung transformiert werden können, damit er prototypisch realisiert werden kann
- und in eine Eiffel-Formulierung, wenn er für das Zielsystem implementiert werden soll.

Die ADL-Formulierung eines Programmbausteins kann jedoch nur in ein Skelett transformiert werden, das der ProST- bzw. der Eiffel-Syntax und Semantik entspricht. Diese Skelette müssen dann vervollständigt werden.

6.5.2 Entwurfsentscheidungen für ADL und ProST

Der Entwurf der Sprachen ADL und ProST wird von folgenden Überlegungen und Randbedingungen geprägt:

- Eiffel ist als Zielsprache vorgegeben. ProST soll auf der Basis von Smalltalk-80 entwickelt werden.
- Die Transformationen nach ProST und nach Eiffel sollen möglichst einfach sein und werkzeugunterstützt durchgeführt werden.
- Die erforderlichen sprachverarbeitenden Werkzeuge für ADL und ProST wie Übersetzer oder Editoren sollen möglichst einfach hergestellt werden können.

Diese Randbedingungen führen dazu, daß der Teil der Sprache ADL, mit dem Programmbausteine formuliert werden, so entworfen ist, daß er sich an der Smalltalk-80-Syntax und -Semantik sowie an der Syntax und Semantik der Sprache Eiffel orientiert. Dadurch werden die Transformationen zwischen den Sprachen einfach, und zumindest ein Teil der sprachverarbeitenden Werkzeuge für ADL sowie die Werkzeuge für ProST können mit geringem Aufwand aus den vorhandenen Smalltalk-80 Werkzeugen hergestellt werden. Diese müssen dazu lediglich angepaßt und erweitert, aber nicht vollständig neu entwickelt werden.

Die folgende Abbildung zeigt, welche Einflüsse die Software-Schemata sowie die Sprachen Eiffel und Smalltalk-80 auf die Konzeption der Sprachen ADL und ProST haben.

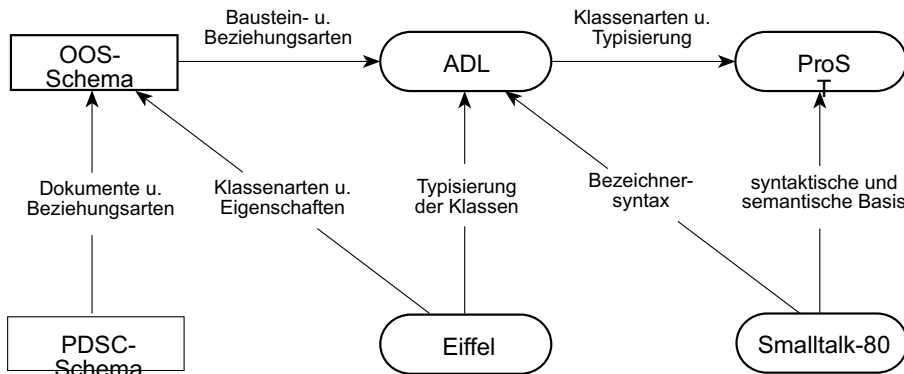


Abb. 6.4: Zusammenhang zwischen Sprachen und Schemata

Das OOS-Schema wurde auf der Basis des PDSC-Schemas entwickelt. Die speziellen objektorientierten Programmbausteine des Schemas entsprechen den Klassenarten, die in Eiffel realisiert werden können.

ADL ist die Sprache, mit der ein Modell gemäß dem OOS-Schema erstellt wird, d.h. dessen Baustein- und Beziehungsarten müssen in ADL formuliert werden können. ADL übernimmt die Bezeichnersyntax von Smalltalk-80 und die Typisierung der Klassen von Eiffel. Dadurch werden die Transformationen nach ProST und Eiffel einfacher.

ProST erweitert Smalltalk-80, damit die in ADL formulierten Klassen auch in ProST notiert werden können. Die Syntax, um die Eigenschaften der Klassen zu definieren, entspricht der ADL-Syntax.

6.5.3 Zur Notation der Syntaxdiagramme

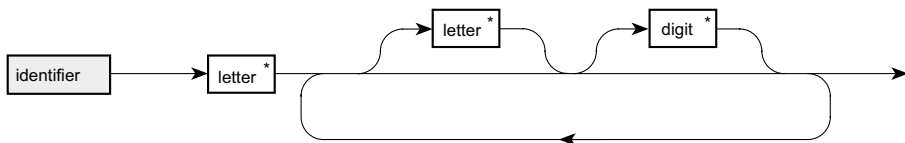
In den Syntaxdiagrammen der Sprachen ADL und ProST werden englische Bezeichner verwendet, damit kein Bruch zur zugrundeliegenden Syntax von Smalltalk-80 entsteht.

Nicht-Terminale, die mit einem „*“ gekennzeichnet sind, sind unverändert aus der Smalltalk-80-Syntax übernommen und werden nicht aufgelöst. Die Syntaxdiagramme selbst werden in der Notation dargestellt, die in Goldberg (1983) verwendet wird.

6.5.4 Bezeichner in ADL und ProST

Die Bezeichnersyntax der Sprachen Smalltalk-80 und Eiffel unterscheidet sich darin, daß Eiffel-Bezeichner Unterstriche enthalten können. Die Menge der Bezeichner, die nach der Smalltalk-80-Syntax gebildet werden, ist dementsprechend eine echte Teilmenge der Bezeichner, die Eiffel akzeptiert.

Aus diesem Grund werden Bezeichner in ADL und ProST – also Bezeichner von Dokumenten, Klassen oder Merkmalen – gemäß der Smalltalk-80-Syntax gebildet. Diese können dann direkt und ohne Transformation bei der Eiffel-Implementierung der Klassen verwendet werden. Das folgende Syntaxdiagramm definiert den Aufbau der Bezeichner für ADL und ProST.



Bezeichner beginnen immer mit einem Buchstaben. Anschließend kann eine Folge, bestehend aus Buchstaben und Ziffern angegeben werden. ADL und ProST unterscheiden bei Bezeichnern nicht zwischen Groß- und Kleinschreibung, d.h. die Bezeichner „adresse“ und „Adresse“ sind identisch.

6.6 Die Architekturbeschreibungssprache ADL

Die Sprache ADL setzt die Baustein- und Beziehungsarten des OOS-Schemas in eine Notation um, mit der Software-Architekturmodelle formuliert werden können. Abstrakt betrachtet, wird mit der Sprache ADL ein Graph beschrieben, dessen Knoten die Dokumente und dessen Kanten die Beziehungen sind.

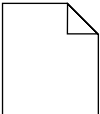



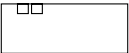



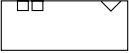



ADL hat eine grafische Syntax, um Dokumente, Klassen und Beziehungen zu formulieren. Sie hat eine lineare Syntax, um die Eigenschaften der Klassen anzugeben und um zusätzliche Informationen zu Dokumenten und Beziehungen zu formulieren.

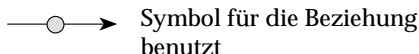
Diese Mischung aus grafischer und linearer Syntax führt auf der einen Seite dazu, daß ein großer Teil des Graphen anschaulich mithilfe von Symbolen angegeben werden kann. Auf der anderen Seite ist die Mischung von grafischer und linearer Syntax jedoch nur akzeptabel, wenn Werkzeuge zur Verfügung gestellt werden, die speziell darauf zugeschnitten sind. Diese Werkzeuge müssen erlauben, daß sowohl die Grafik als auch die textuellen Teile eines Architekturmodells eingegeben und editiert werden können. Sie müssen gewährleisten, daß ein Architekturmodell syntaktisch korrekt formuliert wird.

Beim Entwurf von ADL wurde dieser Werkzeugaspekt explizit miteingeplant. Es ist nicht vorgesehen, daß Software-Architekturmodelle in ADL ohne geeignete Werkzeuge formuliert werden.

6.6.1 Die grafischen Symbole für Dokumente und Beziehungen

Die folgende Tabelle zeigt die Symbole und deren Bedeutung bezogen auf das OOS-Schema, mit denen in ADL Dokumente und deren Beziehungen notiert werden.

	Symbol für ein Dokument		Symbol für die Beziehung steht_in_Beziehung_zu
	Symbol für eine Klasse		Symbol für die Beziehung ist_entstanden_aufgrund
	Symbol für eine generische Klasse		Symbol für die Beziehung ist_Version_von
	Symbol für eine virtuelle Klasse		Symbol für die Beziehung ist_Revision_von
	Symbol für eine virtuelle generische Klasse		Symbol für die Beziehung ist_Variante_zu
			Symbol für die Beziehung ist_Zielimplementierung_zu
			Symbol für die Beziehung erbt_von



6.6.2 Zusätzliche Angaben

Zu jedem Dokument, und damit auch zu jeder Klasse, muß ein Bezeichner angegeben werden. Die Bezeichner aller Dokumente eines Architekturmodells müssen paarweise verschieden sein. Der Bezeichner eines Dokuments wird zusammen mit dem dazugehörenden Symbol abgebildet.

Zusätzlich muß zu jedem Dokument eine erläuternde Beschreibung angegeben werden, die den Zweck und den Inhalt des Dokuments angibt. Sie muß der folgenden Syntax entsprechen:



ADL erlaubt, daß zu jedem Dokument neben dem Bezeichner und einer Beschreibung auch ein Etikett angegeben werden kann, das die Art des Dokuments angibt. Das Etikett kann frei gewählt werden, muß aber der Bezeichnersyntax entsprechen. Da Klassen spezielle Dokumente sind, entfällt die Etikettangabe.

Eine erläuternde Beschreibung muß ebenfalls für Beziehungen der Arten *steht_in_Beziehung_zu*, *ist_entstanden_aufgrund*, *ist_Revision_von*, und *ist_Variante_von* angegeben werden. Zusätzlich zur textuellen Beschreibung kann zu Beziehungen der Art *steht_in_Beziehung_zu* ein Etikett angegeben werden.

6.6.3 Klassen und Eigenschaften

Klassen sowie die Beziehungen zwischen Klassen werden in ADL mit den oben gezeigten grafischen Symbolen deklariert. Es wird eine lineare Syntax verwendet, um die Eigenschaften der Klassen und formale generische Typparameter zu formulieren. Diese wird im folgenden vorgestellt.

Typangaben in ADL

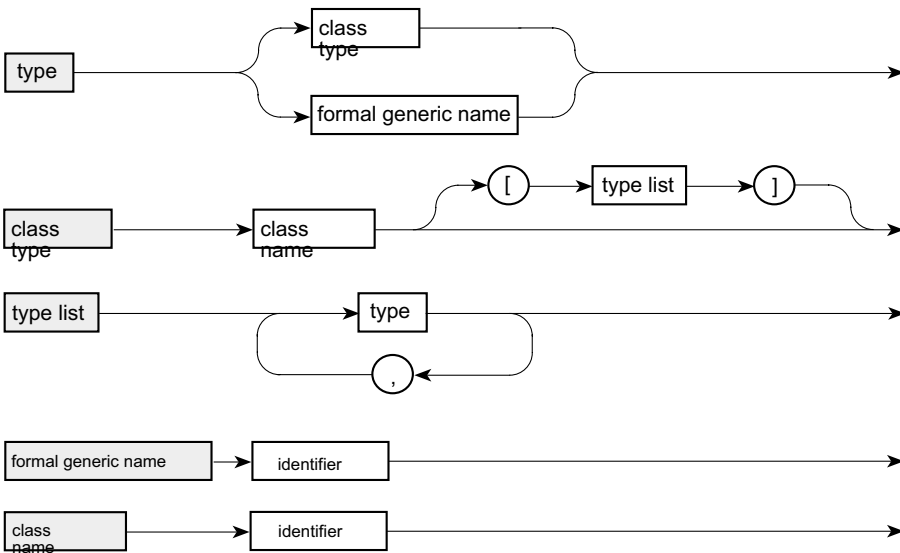
Da Eiffel als Zielsprache eingesetzt wird, werden in ADL die Eigenschaften der Klassen typisiert deklariert. Das bedeutet, daß alle Merkmale und alle Parameter von Routinen mit einer Typangabe versehen werden müssen. Die

typisierte Deklaration der Eigenschaften führt auf der einen Seite dazu, daß eine Klassenbeschreibung in ADL einfach in eine entsprechende Eiffel-Formulierung transformiert werden kann. Auf der anderen Seite ist die Typangabe eine wichtige Information zu Eigenschaften, die explizit angegeben werden soll.

ADL lehnt sich an das Typkonzept der Sprache Eiffel an, so daß drei Arten von Typangaben unterschieden werden:

- Der Bezeichner einer Klasse, die nicht generisch ist, ist ein Typbezeichner.
- Der Bezeichner eines aus einer generischen Klasse instanziierten Typs ist ein Typbezeichner.
- Der Bezeichner eines formalen generischen Typparameters ist ein Typbezeichner.

Die folgenden Syntaxdiagramme beschreiben den Aufbau einer Typangabe.



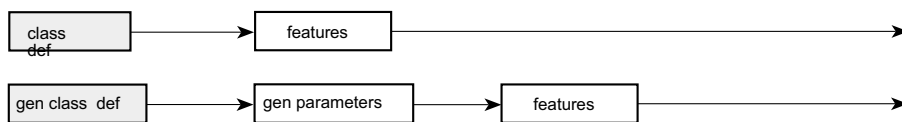
Beispiel:
Set [Point]

Beschreibt eine Menge, deren Elemente Punkte sind.

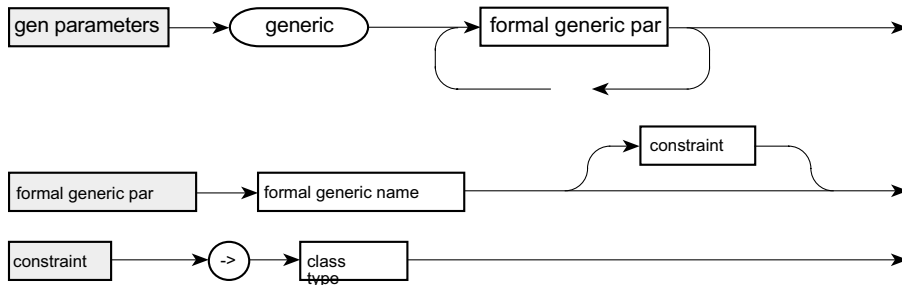
Set[List [Array [Point]]] Beschreibt eine Menge, deren Elemente Listen sind, die selbst Felder von Punkten enthalten.

Syntax der Klassendefinition

Die typisierte Deklaration der Eigenschaften in ADL führt wie in Eiffel dazu, daß für generische Klassen zusätzlich zu den Eigenschaften immer auch formale generische Typparameter angegeben werden müssen. Dies wird syntaktisch berücksichtigt.



Formale generische Typparameter werden mit dem Schlüsselwort „generic“ eingeleitet und in der Eiffel-üblichen Notation formuliert. Da Eiffel die uneingeschränkte und die auf einen Typ und seine Untertypen eingeschränkte Generizität erlaubt, muß dies bereits bei den formalen generischen Parametern angegeben werden können.



Beispiel:

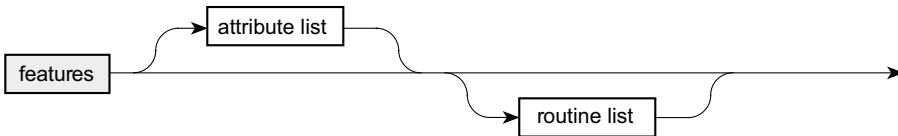
```

generic
  T,
  Geo -> GeoObject
  
```

Führt die Bezeichner „T“ und „Geo“ als formale generische Typparameter ein. Es können generisch instanziierte Typen erzeugt werden, wobei als aktueller Parameter für „T“ jeder beliebige Typ, für „Geo“ aber nur Typen angegeben werden können, die konform zum Typ „GeoObject“ sind.

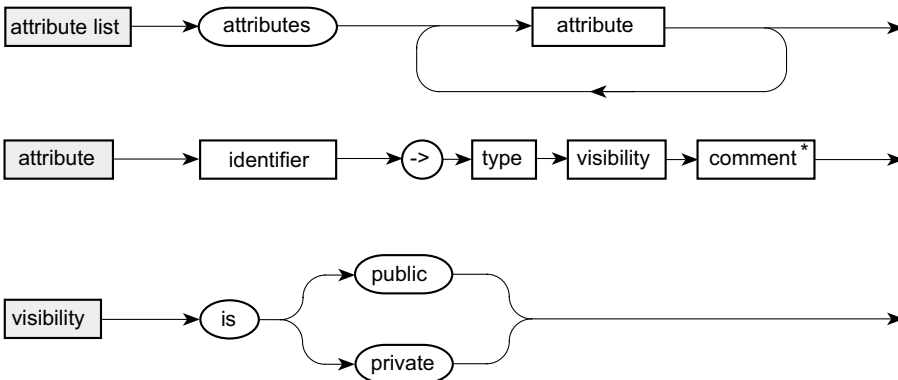
Deklaration der Eigenschaften

Die Eigenschaften einer Klasse werden getrennt nach Art angegeben: zuerst die Liste der Merkmale, dann die Liste der Routinen. Beide Listen können leer sein.



Merkmale

Die Liste der Merkmale wird mit dem Schlüsselwort „attributes“ eingeleitet. Zu jedem Merkmal muß ein Bezeichner, ein Typ, die Sichtbarkeit des Merkmals (öffentlich oder privat) und ein erläuternder Kommentar angegeben werden.



Beispiel:

Nachfolgend sind Merkmale der Klasse „Rechteck“ in ADL formuliert. Ein Rechteck wird durch vier Merkmale beschrieben, die die Eckpunkte definieren. Zwei Merkmale – „linksOben“ und „rechtsUnten“ – sind öffentlich. Die anderen zwei sind privat. Alle Merkmale sind vom Typ Punkt.

```

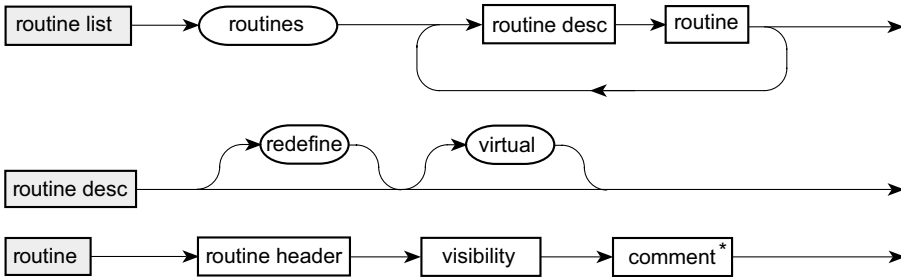
attributes
  origin -> Point is public
    „top left corner“
  corner -> Point is public
    „bottom right corner“
  bottomLeft -> Point is private
    „bottom left corner“

```

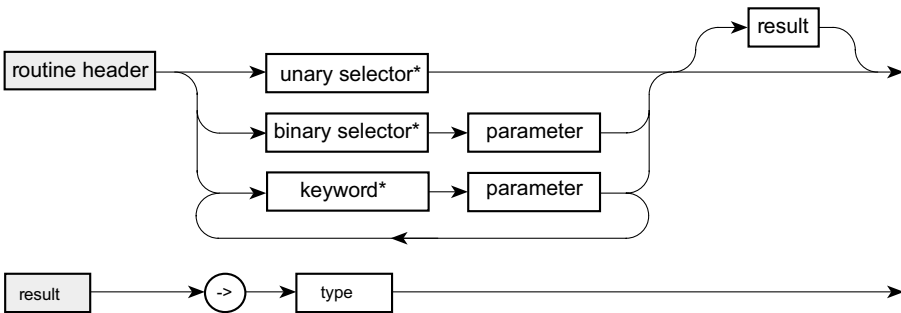
```
topRight -> Point is private  
„bottom right corner“
```

Routinen

Die Liste der Routinen wird mit dem Schlüsselwort „routines“ eingeleitet. Eine Deklaration legt den Bezeichner der Routine sowie eventuelle Parameter fest. Eine virtuelle Routine wird deklariert, wenn das Schlüsselwort „virtual“ vor den Bezeichner der Routine gestellt wird. Eine geerbte Routine, die überdefiniert werden soll, wird mit dem Schlüsselwort „redefine“ markiert.



Die Syntax, um Bezeichner für Routinen zu bilden, entspricht der Smalltalk-80 Syntax. Es können somit einstellige Routinen, die keinen Parameter haben, zweistellige Routinen, die genau einen Parameter haben und in Infix-Notation geschrieben werden, sowie Routinen in der Schlüsselwortnotation formuliert werden.



Die Deklaration eines formalen Parameters besteht aus einem Bezeichner, aus einem Typ und wird von runden Klammern begrenzt.



Beispiel:

Nachfolgend sind einige Routinendeklarationen der Klasse „Rechteck“ aufgeführt.

```
routines
  contains: (p : Point) -> Boolean is public
    „answers whether point p is inside“
  origin: (l : Point) corner: (r : Point) is public
    „sets origin to l and corner to r“
  center -> Point is public
    „answers the center point“
```

6.7 ProST: eine Smalltalk-80 basierte Prototyping-Sprache

In diesem Abschnitt wird die Prototyping-Sprache ProST vorgestellt. Es werden die Konstrukte der Sprache ProST im einzelnen eingeführt, die über die Sprachdefinition von Smalltalk-80 hinausgehen. Smalltalk-80 wird in dieser Arbeit nicht erläutert. Sie wird beispielsweise in Goldberg (1983) vollständig vorgestellt.

6.7.1 Vorbemerkungen

Die Smalltalk-80-Erweiterung ProST muß die nachstehenden Anforderungen erfüllen:

- Alle Programmbausteinarten, die das OOS-Schema definiert, müssen in ProST formuliert werden können.
- ProST muß wie Smalltalk-80 für das Prototyping geeignet sein. Der Charakter von Smalltalk-80 muß erhalten bleiben, damit diese Eigenschaft nicht verloren geht.

Diese Anforderungen führen zu folgender Entwurfsentscheidung: ProST verlangt, daß Typangaben gemacht werden müssen. Es besteht aber weiterhin kein Deklarationszwang für lokale Variablen, die eine Routine in ihrem Rumpf verwendet. Eine Typprüfung findet nicht statt.

ProST geht in folgenden Punkten über die Sprachdefinition von Smalltalk-80 hinaus:

- Es können generische Klassen definiert werden und Objekte davon erzeugt werden.
- Virtuelle Klassen können definiert werden.

- Merkmale (Instanzvariablen) und die formalen Parameter der Routinen müssen typisiert deklariert werden.

Um dies zu realisieren, wird die Smalltalk-80-Syntax verändert. Weiterhin werden spezielle Systemklassen zur Verfügung gestellt, damit ProST-Klassen erzeugt werden können¹.

Nichtterminale, die in den nachfolgend vorgestellten Produktionen der ProST-Syntax auftreten und mit Nichtterminalen der ADL-Syntax übereinstimmen, sind mit „^“ markiert. Sie werden in der Sprachbeschreibung von ADL aufgelöst.

6.7.2 Klassen in ProST

Smalltalk-80 hat keine eigene Syntax, um Klassen zu definieren. Dies geschieht, indem der gewählten Oberklasse eine entsprechende Nachricht geschickt wird. Diese Nachricht ist mit dem Bezeichner der neuen Klasse und mit den Bezeichnern der Instanz-, Klassen- und Poolvariablen parametrisiert.

Beispiel:

Soll eine Klasse `Box` mit den drei Instanzvariablen `container`, `numberOfCards` und `selectedCard` als Unterklasse der Klasse `Object` deklariert werden, so geschieht dies in Smalltalk-80 mit folgender Nachricht:

```
Object subclass: #Box
  instanceVariables: `container
                    numberOfCards
                    selectedCard`
  classVariables: ``
  poolVariables: ``
  categorie: `Bibliography`
```

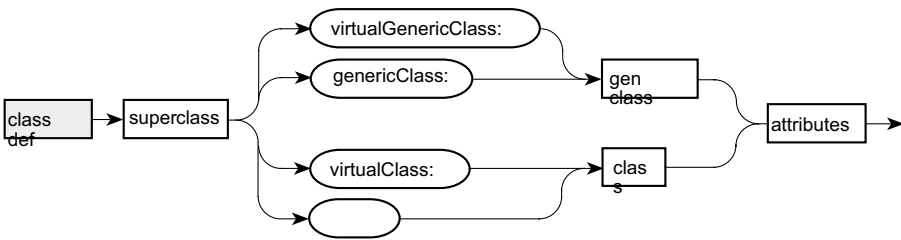
Diese Nachricht kann nicht mehr verwendet werden, um ProST-Klassen zu deklarieren, da für diese andere Informationen benötigt werden. Die entsprechende ProST-Formulierung sieht folgendermaßen aus:

```
Object genericClass: `Box[Card]`
  attributes: `(container : Array [Card])
              (numberOfCards : Integer)
              (selectedCard : Card)`
```

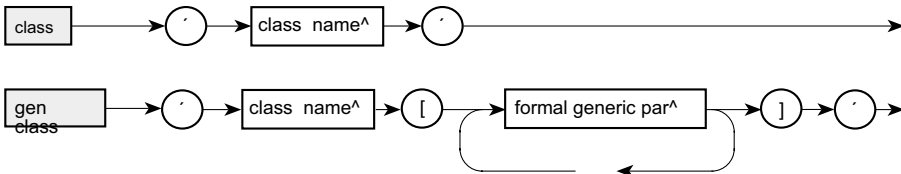
¹ Siehe dazu auch Abschnitt 7.4.1

Das Beispiel zeigt, wie eine generische Klasse in ProST deklariert wird: „Card“ ist der formale generische Typparameter; alle Attribute haben eine Typangabe. Das Attribut „container“ ist beispielsweise ein Feld, dessen Elemente Karteikarten sind.

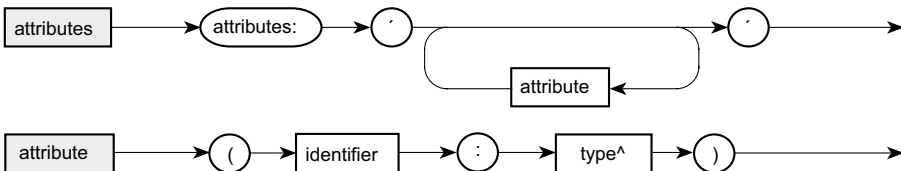
ProST-Klassen werden analog zu Smalltalk-80-Klassen deklariert, indem spezielle Nachrichten an die gewählte Oberklasse geschickt werden. Für jede der vier möglichen Klassenarten, die das OOS-Schema definiert, ist eine eigene Nachricht vorgesehen.



Der Bezeichner einer Klasse sowie die eventuell vorhandenen formalen generischen Typparameter, die gemäß der Eiffel-Syntax angegeben werden, werden in Hochkommata eingeschlossen.



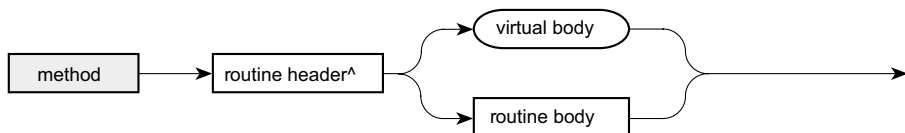
Bei der Deklaration einer ProST-Klasse werden die Merkmale angegeben, die ihre Objekte charakterisieren. Die Liste der Merkmale wird ebenfalls in Hochkommata eingeschlossen. Jedes Merkmal besteht aus einem Bezeichner, aus einer Typangabe und wird von runden Klammern begrenzt.



Typangaben werden in ProST entsprechend der ADL-Syntax vorgenommen.

6.7.3 Routinen in ProST

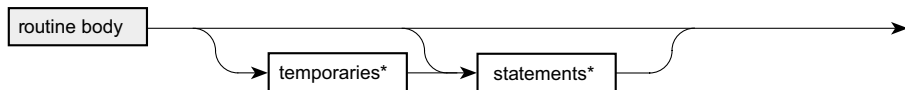
Routinen werden in ProST gemäß der ADL-Syntax deklariert (siehe Produktion „routine header“). Um Routinen und insbesondere virtuelle Routinen in ProST zu formulieren, wird die Produktion „method“ der Smalltalk-80-Syntax durch die folgende veränderte Produktion ersetzt.



Für virtuelle Routinen definiert ProST einen vorgegebenen Rumpf. Dieser besteht einzig aus der in Smalltalk-80 zu diesem Zweck definierten Nachricht „subclassResponsibility“.



Im Rumpf einer nicht-virtuellen Routine können Hilfsvariablen deklariert werden und Ausdrücke gemäß der Smalltalk-80-Syntax stehen.



Kapitel 7

Die Prototyping-Werkstatt ProWork

In der Werkstatt *ProWork* (**Prototyping Workshop**) sind alle Werkzeuge zusammengefaßt, mit denen entsprechend der Vorgehensweise PDSC Architekturmodelle objektorientierter Anwendungssoftware erstellt, Architekturprototypen konstruiert und schrittweise in das Zielsystem überführt werden können. Dabei werden die Sprachen ADL, ProST und Eiffel eingesetzt.

In diesem Kapitel werden die wichtigsten Werkzeuge, ihr Zusammenspiel, ihre Funktionalität und relevante Implementierungsdetails beschrieben.

7.1 Werkzeug und Werkstatt

Budde und Züllighoven (1990) führen in ihrer Arbeit eine detaillierte Diskussion über die Begriffe Werkzeug, Werkstatt und Fabrik. Dabei formulieren sie auf den Seiten 201 bis 215 eine Liste von Anforderungen, die die Handhabung und die Gestaltung von Werkzeugen – damit sind immer Software-Werkzeuge gemeint – betreffen. Die in diesem Kapitel vorgestellten Werkzeuge sind entsprechend diesen Anforderungen konstruiert, soweit sie für die einzelnen Werkzeuge relevant sind. Eine Werkstatt wird in der zitierten Arbeit auf Seite 219 folgendermaßen definiert: „Eine Programmierumgebung als *Werkstatt* stellt Werkzeuge zur Verfügung und besitzt keine Gesamtstrategie der Software-Entwicklung. Sie unterscheidet sich von einer Programmierumgebung als *Fabrik* dadurch, daß diese eine durchgängige, vorprogrammierte Strategie implementiert, um Software zu entwickeln“.

ProWork ist in diesem Sinn eine Programmierumgebung, die den Charakter einer Werkstatt hat. Obwohl es mit PDSC eine Vorgehensweise gibt, um einen Architekturprototyp in das Zielprogramm zu überführen, ist die Software-Entwicklung mit den Werkzeugen von ProWork nicht vorpro-

grammiert oder sogar vollständig automatisiert. Es Bedarf weiterhin der Intuition und Erfahrung der Entwickler, um mithilfe der bereitgestellten Werkzeuge von der Problemstellung zur fertigen Software zu gelangen.

7.2 Die Werkstatt im Überblick

In Abschnitt 4.6 werden in allgemeiner Form eine Werkstatt und deren Werkzeuge vorgestellt, um PDSC zu unterstützen. Die Werkzeuge der Werkstatt ProWork sind entsprechend dem dort vorgestellten Konzept entwickelt.

7.2.1 Die angebotenen Werkzeuge

In ProWork sind die in Abbildung 7.1 gezeigten Werkzeuge zusammengefaßt. Die Werkzeuge sind in Werkzeugkästen gruppiert:

- Der *Architekturwerkzeugkasten* enthält den ADL-Architktureditor, den ADL-Texteditor, den ADL-Compiler sowie die beiden Transformatoren ADL-ProST und ADL-Eiffel.
- Im *Zielsprachenwerkzeugkasten* sind alle Werkzeuge enthalten, die für die Programmierung in Eiffel zur Verfügung gestellt werden. Diese werden in Eiffel (1990) beschrieben. Ferner enthält dieser Werkzeugkasten einen Texteditor, um Eiffel-Code eingeben und ändern zu können.
- Der *Prototyping-Werkzeugkasten* enthält die ProST-spezifischen Werkzeuge ProST-Compiler und ProST-Browser sowie einen Transformator, um den ProST-Code einer Klasse so zu verändern, daß deren Eiffel-Implementierung verwendet wird (ProST-Eiffel-I-Transformator). Zusätzlich enthält der Werkzeugkasten die Werkzeuge der Smalltalk-80 Programmierumgebung. Dazu zählt z.B. der Objektinspektor oder der Debugger. Goldberg (1984) beschreibt, was die Werkzeuge der Smalltalk-80 Programmierumgebung leisten und wie sie zu bedienen sind. Sie werden deshalb nicht weiter erläutert und sind in der Abbildung 7.1 auch nicht im einzelnen dargestellt.
- Der *Ausführungswerkzeugkasten* enthält die Werkzeuge, die notwendig sind, um einen Architekturprototyp, der aus ProST- und Eiffel-Komponenten besteht, zusammenzubinden und auszuführen.

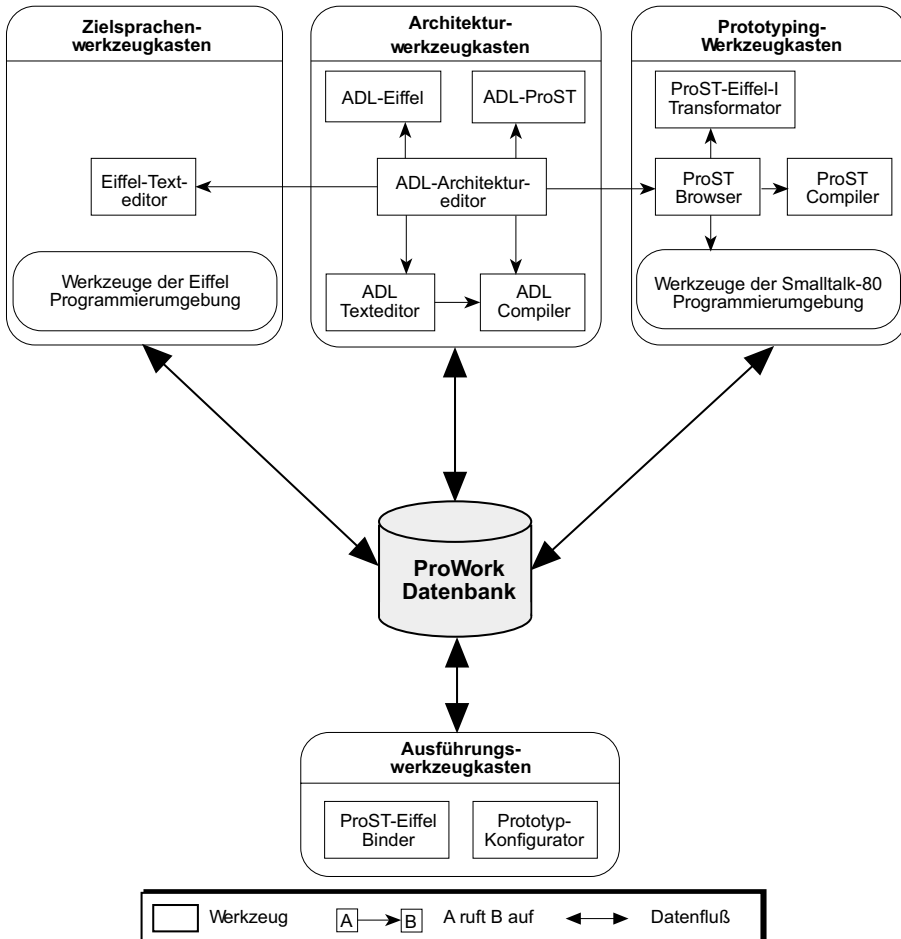


Abb. 7.1: Die Werkstatt ProWork im Überblick

7.2.2 Der Gebrauch der Werkzeuge

Alle Werkzeuge arbeiten, soweit erforderlich, auf der zentralen Datenerhaltungskomponente der Werkstatt – *ProWork-Datenbank*. Diese wird in Abschnitt 7.5 beschrieben.

Das zentrale Werkzeug der Werkstatt ist der *ADL-Architektureditor*. Mit ihm werden die Architekturmodelle – teils in interaktiver grafischer Art, teils textuell – formuliert und modifiziert. Zu diesem Zweck kann aus dem *ADL-Architektureditor* der *ADL-Texteditor* aufgerufen werden. Der *ADL-*

Architektureditor und der ADL-Texteditor rufen den *ADL-Compiler* auf, um die textuellen Angaben eines Architekturmodells zu überprüfen.

Soll eine Klasse prototypisch in ProST realisiert werden, so wird aus dem Architektureditor der *ADL-ProST-Transformer* aufgerufen. Dieser erzeugt aus der ADL-Formulierung das Skelett einer entsprechenden ProST-Klasse. Mit dem *ProST-Browser* werden die erzeugten Klassenrahmen betrachtet und die deklarierten Routinen der Klassen implementiert. Der ProST-Browser verwendet den *ProST-Compiler*, um die eingegeben Routinen-Rümpfe zu übersetzen.

Soll eine Klasse für das Zielprogramm in Eiffel implementiert werden, wird aus der ADL-Formulierung mithilfe des *ADL-Eiffel-Transformators* das Skelett einer entsprechenden Eiffel-Klasse erzeugt. Dieses muß mit einem Texteditor weiterbearbeitet werden, bis die Implementierung abgeschlossen und die Klasse getestet ist.

Um eine Klasse, die bereits in Eiffel implementiert ist, in den Architekturprototyp zu integrieren, werden zwei Werkzeuge zur Verfügung gestellt:

- Der *ProST-Eiffel-I-Transformer* verändert den vorhandenen ProST-Code der Klasse, damit im Architekturprototyp anstatt der ProST-Implementierung die angebundene Eiffel-Implementierung verwendet wird.
- Der *ProST-Eiffel-Binder* wird verwendet, um den Eiffel-Code letztlich an den ProST-Code zu binden.

In Abschnitt 7.6 wird detailliert beschrieben, wie Eiffel an ProST angebunden wird.

Mit dem Prototypkonfigurator wird der Architekturprototyp konfiguriert, bevor er ausgeführt werden kann. Der Konfigurator berücksichtigt, daß Klassen eventuell in Revisionen und Varianten vorliegen; er wird in Abschnitt 7.7.1 vorgestellt.

7.2.3 Anmerkungen zur Realisierung

Die Mehrzahl der Werkzeuge ist in Smalltalk-80 realisiert. Dies hat folgende Gründe:

- Da die Sprachen ADL und ProST auf Smalltalk-80 basieren, können ein Teil der vorhandenen Smalltalk-80 Werkzeuge unverändert verwendet werden. Dazu zählen z.B. der Debugger oder der Objektinspektor. Andere Werkzeuge wie ADL- und ProST-Compiler konnten, nachdem

sie an die Sprachen ADL und ProST angepaßt sind, in ProWork zur Verfügung gestellt werden. Dadurch wurde der Aufwand erheblich verringert, um die Werkzeuge zu realisieren.

- Es sind Erfahrungen bei der Entwicklung interaktiver Anwendungen mit Smalltalk-80 vorhanden gewesen, die genutzt werden konnten.

Die Werkzeuge, die notwendig sind, um einen Architekturprototyp zusammenzubinden, der sowohl aus ProST- als auch aus Eiffel-Komponenten besteht, sind in der Sprache C oder als UNIX-Kommandodateien realisiert. Dies war notwendig, weil vorhandene betriebssystemnahe Werkzeuge wie der C-Compiler oder das UNIX-Werkzeug „make“ verwendet wurden.

Die Werkzeuge von ProWork wurden auf einer Risc-Workstation der Firma DEC unter dem Betriebssystem ULTRIX entwickelt.

7.3 Der Architektureditor für ADL

Die Software-Modellierungssprache ADL definiert Symbole, um Dokumente und Klassen sowie deren Beziehungen in einem Architekturmodell zu formulieren. Sie hat eine lineare Syntax, um die Eigenschaften sowie zusätzliche Angaben zu Dokumenten und Beziehungen anzugeben. Der ADL-Architektureditor ist auf die Mischung von grafischer und linearer Syntax angepaßt.

Er hat folgende wesentliche Funktionsmerkmale:

- Ein Architekturmodell kann grafisch mit den Symbolen der Sprache ADL eingegeben und interaktiv bearbeitet werden. Der Editor erlaubt nur solche Architekturmodelle, die dem OOS-Schema entsprechen. Die dort definierten Konsistenzbedingungen werden überprüft.
- Er bietet die Möglichkeit, noch vage Modellelemente entsprechend den im OOS-Schema definierten Möglichkeiten zu spezialisieren. So kann z.B. eine Klasse zu einer virtuellen oder generischen Klasse spezialisiert werden.
- Er speichert ein Architekturmodell in der zentralen ProWork-Datenbank. Gespeicherte Modelle können wieder eingelesen werden.
- Andere Werkzeuge sind an ihn angeschlossen. So wird der ADL-Texteditor aufgerufen, wenn Teile des Architekturmodells anzugeben sind, die nicht grafisch, sondern gemäß der linearen ADL-Syntax formuliert werden müssen. Die beiden Transformatoren, ADL-ProST

und ADL-Eiffel, können ebenfalls direkt aus dem Architektureditor aufgerufen werden.

Der ADL-Architektureditor ist mit dem „Application Framework“ vis-A-vis realisiert. Es erlaubt, grafische Entwurfswerkzeuge schnell und mit geringem Programmieraufwand – im Vergleich zu einer völlig neuen Implementierung – zu entwickeln. vis-A-vis wurde in der Abteilung Software-Engineering des Instituts für Informatik an der Universität Stuttgart entwickelt. Es wird im nächsten Abschnitt vorgestellt.

7.3.1 Das „Application Framework vis-A-vis“

Ein „Application Framework“ besteht nach Bischofberger (1990) aus einer objektorientiert konstruierten Klassenbibliothek und aus einer Standardanwendung, die erweitert und angepaßt werden kann. vis-A-vis ist speziell geeignet, um interaktive Editoren für grafische Notationen zu erstellen. vis-A-vis stellt dazu einen „offenen Basiseditor“, zentrale Mechanismen und wiederverwendbare Klassen in Form einer Bibliothek zur Verfügung, mit der schnell und auf einheitliche Weise ein Editor für eine spezielle grafische Notation entwickelt werden kann.

Mit vis-A-vis werden grafische Notationen unterstützt, die aus Symbolen für Objekte und Relationen bestehen. vis-A-vis erlaubt, daß sowohl auf einzelnen semantischen Elementen des Modells (also auf den Objekten und Relationen) als auch auf dem gesamten Modell Operationen ausgeführt werden können.

vis-A-vis eignet sich nicht, um Werkzeuge zu implementieren, die außerhalb seiner Anwendungsklasse liegen. Darin besteht der Hauptunterschied zu mächtigeren Application Frameworks wie ET++, das in Gamma (1988) und Weinand (1989) beschrieben ist. vis-A-vis ist auch nicht darauf ausgelegt, um beliebige Benutzungsschnittstellen zu konstruieren, wie dies etwa das in Böcker (1992) vorgestellte System VICK erlaubt.

vis-A-vis läßt sich zusammenfassend folgendermaßen charakterisieren:

- Es ist relativ klein und kann schnell erlernt und eingesetzt werden. Die vis-A-vis Klassenbibliothek umfaßt rund 60 Klassen, die teilweise direkt verwendet werden können, um spezielle Editoren zu bauen. Ein Großteil der Klassen bildet jedoch eine Basis, von der geerbt wird, wenn für einen Editor neue Klassen entwickelt werden müssen.

- vis-A-vis definiert eine Standardarchitektur für alle damit erzeugten Editoren. Diese Architektur wird hauptsächlich von den folgenden Aspekten beeinflusst:
 - Die in Budde (1992a) beschriebene „Werkzeug-Material-Metapher“, die angibt, wie interaktive Werkzeuge intern aufgebaut sein sollen.
 - Die strikte Trennung zwischen grafischer Notation (Symbole für Objekte und Relationen) und semantischem Modell.
 - Der Aufbau einer Benutzungsschnittstelle nach der „Model-View-Controller“-Konstruktion, die in Krasner (1988) beschrieben ist.
- vis-A-vis definiert ein einheitliches Aussehen für alle Editoren und legt fest, wie die Werkzeuge zu bedienen sind. Dies führt auf der einen Seite dazu, daß der Werkzeugbauer nur noch eingeschränkte Möglichkeiten hat, um die Benutzungsschnittstelle zu gestalten. Auf der anderen Seite kann ein Anwender, der gelernt hat, mit einem vis-A-vis-Editor zu arbeiten, mit allen vis-A-vis Editoren arbeiten.
- Die allgemeinen Mechanismen, die für jeden Editor benötigt werden, z.B. Speichern und Laden der Modelle, werden bereits fertig zur Verfügung gestellt.
- Zu vis-A-vis ist eine Vorgehensweise formuliert, die beschreibt, wie auf systematische Art und Weise Editoren mit vis-A-vis entwickelt werden. Dazu werden Metawerkzeuge angeboten, die der Werkzeugbauer verwenden kann.

In Lichter (1993) ist vis-A-vis eingehend beschrieben. Insbesondere wird dort erläutert, welche Anforderungen an die Konstruktion von vis-A-vis gestellt wurden, wie vis-A-vis-Editoren intern aufgebaut sind und wie spezielle Editoren mit vis-A-vis konstruiert werden.

7.3.2 Der Architektureditor

Abbildung 7.2 ist ein Bildschirmabzug, auf dem ein ADL-Architektureditor zu sehen ist. Der Editor ist entsprechend der vis-A-vis-Vorgabe aus vier Regionen aufgebaut: In der *Palette* werden die Symbole der Sprache ADL angeboten, mit denen Architekturmodelle formuliert werden können. In der *Menüleiste* werden die Funktionen angeboten, mit denen das gesamte Modell bearbeitet werden kann, z.B. Speichern und Laden des Modells. Im *Zeichenfenster* werden die Elemente des Architekturmodells plaziert. Sie können bewegt und beliebig auf der Zeichenfläche angeordnet werden. Im *Nachrichtenfenster* werden Fehlermeldungen und Hilfetexte angezeigt.

Mit der Maus kann zu jedem Modellelement ein spezielles Menü erhalten werden, das die Funktionen anbietet, die zu der entsprechenden Elementart vorgesehen sind. In Abbildung 7.2 ist ein solches Menü für Klassen zu sehen. Mit diesem Menü können beispielsweise die beiden Transformatoren ADL-ProST und ADL-Eiffel gestartet sowie die entsprechenden Editoren für ADL, ProST und Eiffel aufgerufen werden.

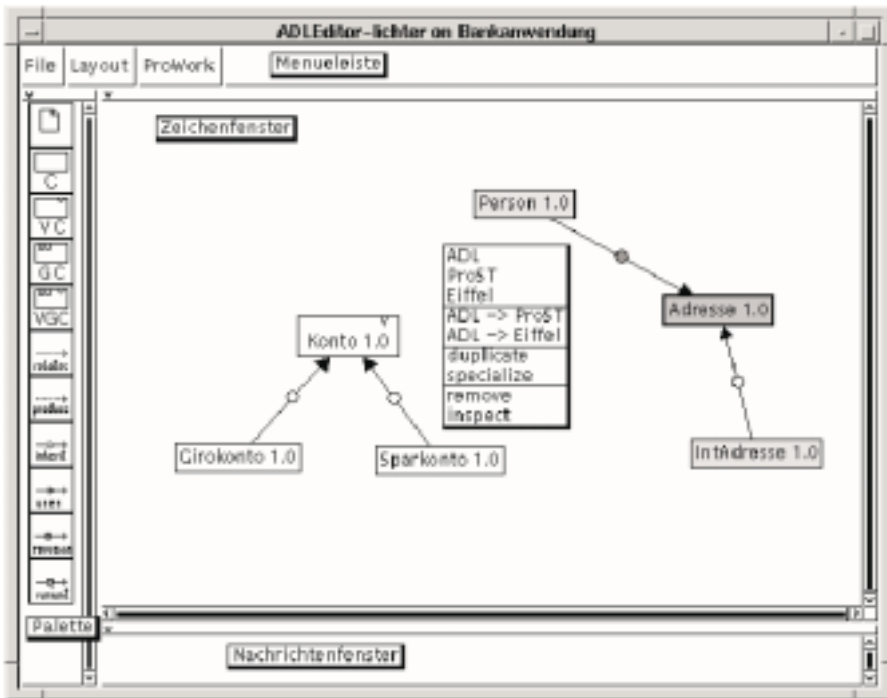


Abb. 7.2: Der ADL-Architektureditor

7.4 Die ProST-Sprachwerkzeuge

ProWork stellt spezielle Werkzeuge zur Verfügung, um modellierte Programmbausteine prototypisch in ProST zu realisieren. Im folgenden werden diese Werkzeuge vorgestellt, und es wird erläutert, wie ProST in die Smalltalk-80 Programmierumgebung eingebettet ist.

7.4.1 Die Integration von ProST in die Smalltalk-80 Programmierumgebung

Da ProST eine Spracherweiterung von Smalltalk-80 ist, ist die Sprache so in die Smalltalk-80 Programmierumgebung eingebettet, daß

- ProST-Klassen mit speziellen Werkzeugen erstellt und modifiziert werden können,
- in ProST-Routinen die vorhandenen Smalltalk-80-Nachrichten verwendet werden können,
- weiterhin in Smalltalk-80 implementiert werden kann.

Um dieses zu erreichen, wurde die Programmierumgebung im wesentlichen in folgenden Punkten erweitert:

- Es wurden zwei sprachspezifische Werkzeuge entwickelt, der ProST-Compiler und der ProST-Browser.
- Der Smalltalk-80 Klassenbaum wurde um einen ProST-spezifischen Zweig erweitert. Dieser Zweig beginnt bei der Klasse *ProSTRoot*. Sie ist eine direkte Unterklasse der Wurzelklasse „Object“ und hat selbst zwei Unterklassen: *ProSTObject* und *ProSTGenObject*.

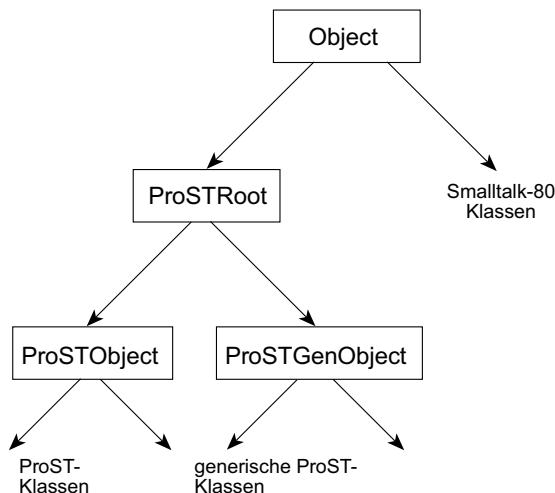


Abb. 7.3: Integration von ProST in die Smalltalk-80 Programmierumgebung

Die neuen Klassen dienen zu folgenden Zwecken:

- *ProSTRoot* legt fest, daß alle Unterklassen mit dem ProST-Compiler übersetzt werden.
- *ProSTObject* ist die Wurzelklasse aller nicht-generischen ProST-Klassen. Sie definiert spezielle Methoden, um ProST-Klassen zu erzeugen.
- *ProSTGenRoot* ist die Wurzelklasse aller generischen ProST-Klassen. Sie definiert Methoden, um generische ProST-Klassen zu erzeugen.

7.4.2 ProST-Browser und -Compiler

Der ProST-Browser ist, analog zum Smalltalk-80-Browser, das zentrale Werkzeug, um ProST-Klassen zu betrachten, Code einzugeben und um Informationen über Klassen zu erhalten.

Abbildung 7.4 ist der Bildschirmabzug eines ProST-Browsers, der den vom ADL-ProST-Transformator erzeugten Code der Klasse „Adresse“ zeigt. Das Menü bietet neben den für ein solches Werkzeug üblichen Informationsmöglichkeiten Funktionen an, um den ProST-Code einer Klasse in der ProWork-Datenbank zu speichern oder gespeicherten Code von dort zu lesen. Weiterhin kann über dieses Menü der ProST-Eiffel-I-Transformator gestartet werden, wenn die Eiffel-Implementierung der Klasse im Architekturprototyp verwendet werden soll.

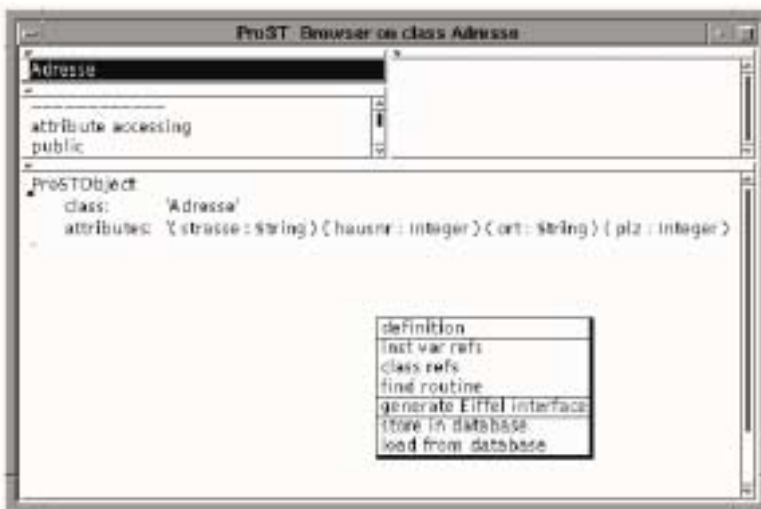


Abb. 7.4: Der ProST-Browser

Der im ProST-Browser eingegebene Code wird inkrementell mit dem ProST-Compiler übersetzt. Dieser wurde auf der Basis des existierenden Smalltalk-80-Compilers entwickelt. Dazu wurde der ProST-Parser als Spezialisierung des Smalltalk-80-Parsers und der ProST-Compiler als Spezialisierung des Smalltalk-80-Compilers in die Programmierumgebung eingebracht. Die objektorientierte Konstruktion des vorhandenen Smalltalk-80-Compilers hat den Aufwand erheblich reduziert, um ProST-Parser und -Compiler zu entwickeln, da große Teile geerbt oder in angepaßter Form wiederverwendet werden konnten.

7.5 Die Datenhaltung

Das Herz der Werkstatt ProWork ist eine Datenbank, die alle modellierten Dokumente und Beziehungen speichert und verwaltet. Diese Datenbank ist mit dem objektorientierten Datenbanksystem GemStone entwickelt und in dessen Sprache OPAL formuliert. GemStone wurde gewählt, weil

- es eine gute und benutzungsfreundliche Schnittstelle zur Smalltalk-80 Programmierumgebung hat und
- die objektorientierten Datenmodellierungsmöglichkeiten geeignet sind, um die im OOS-Schema definierten Baustein- und Beziehungsarten in einem Datenbankschema zu formulieren. Baustein- und Beziehungsarten werden durch entsprechende OPAL-Klassen im Datenbankschema beschrieben. Die Spezialisierungsbeziehungen zwischen Baustein- oder Beziehungsarten werden mithilfe der Vererbung im Datenbankschema formuliert.

Die Modellierungskonzepte, die objektorientierte Datenbanksysteme – und damit auch GemStone – anbieten, um Datenbankschemata zu definieren, werden beispielsweise in Heuer (1992) beschrieben. GemStone und seine Programmiersprache OPAL werden in Servio (1991) detailliert vorgestellt. Beide Aspekte werden in nächsten Abschnitt nur kurz diskutiert.

7.5.1 GemStone - Ein Überblick

Ein Datenbankmetaschema beschreibt, welche Möglichkeiten ein Datenbanksystem zur Verfügung stellt, um Datenbankschemata zu definieren. Das Datenbankmetaschema von GemStone ist objektorientiert. Es sieht verkürzt formuliert vor, daß ein Datenbankschema durch Klassen definiert wird, die mit der Einfachvererbung strukturiert sind. Klassen werden in OPAL formuliert. Es können Instanz- und Klassenvariablen sowie Instanz- und Klassenmethoden angegeben werden.

Alle Objekte, die von einer Schemaklasse erzeugt werden, sind persistent. GemStone bietet mit sogenannten Behälterobjekten eine Möglichkeit, um Objekte zusammenzufassen, damit ihnen auf einfache Art Anfrage- und Aktualisierungsnachrichten geschickt werden können. Zu diesem Zweck sind allgemeine und häufig gebrauchte Behälterklassen in der GemStone-Klassenbibliothek vordefiniert.

7.5.2 Die ProWork-Datenbank

GemStone erlaubt, eigene Behälterklassen zu definieren, die besonders für die beabsichtigte Datenhaltung geeignet sind. Dazu kann von den bereits vorhandenen Behälterklassen geerbt werden. Von dieser Möglichkeit wurde bei der Entwicklung der ProWork-Datenbank Gebrauch gemacht, da diese speziell Dokumente und Beziehungen zwischen Dokumenten speichern und verwalten soll.

Die Klasse *ProWorkDB* wurde als Datenbehälterklasse in die GemStone Klassenbibliothek eingebracht. Eine ProWork-Datenbank ist ein Objekt dieser GemStone-Klasse. Die Klasse „ProWorkDB“ ist so konzipiert, daß alle Werkzeuge auf den gespeicherten Daten arbeiten können. Zu diesem Zweck werden alle Daten – also Dokumente und Beziehungen – zusätzlich zu ihrer Objektidentität mit einem eindeutigen Schlüssel versehen, der von den Werkzeugen, die außerhalb der Datenbank arbeiten, verwendet wird. Diese Schlüssel werden von der Datenbank nach dem Surrogatkonzept vergeben.

Damit die Werkzeuge von ProWork auf Datenänderungen in der Datenbank reagieren können, melden sie sich bei der Datenbank an. Dadurch ist es möglich, alle auf der Datenbank arbeitenden Werkzeuge bei Datenänderungen zu aktualisieren. Die nachfolgende Abbildung zeigt die wesentliche interne Struktur einer ProWork-Datenbank.

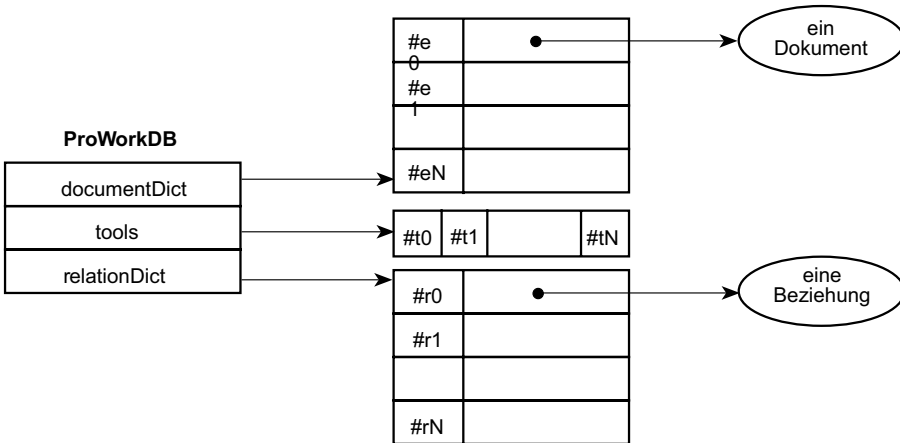


Abb. 7.5: Der interne Aufbau einer ProWork-Datenbank

Eine ProWork-Datenbank enthält zwei Behälterobjekte „documentDict“ und „relationDict“, in denen die Objekte der Dokument- und der Beziehungsklassen gespeichert werden. Weil diese Behälterobjekte Diktionäre sind, werden die einzelnen Einträge in Form von Assoziationen gespeichert. Der Schlüssel einer Assoziation ist ein Surrogat, der Wert das eigentlich zu speichernde Objekt. Im Behälterobjekt „tools“ werden alle aktuell auf einer Datenbank arbeitenden Werkzeuge in Form eines eindeutigen Werkzeugschlüssels registriert.

Mithilfe von Anfrageoperationen, die die Klasse ProWorkDB zur Verfügung stellt, kann ein Werkzeug selektiv eine Menge von Dokumenten und Beziehungen bearbeiten. Die Konsistenzbedingungen, die das OOS-Schema definiert, werden durch entsprechende Operationen überprüft.

7.5.3 Das Datenbankschema

Die nachstehende Abbildung zeigt die wichtigsten Klassen des ProWork-Datenbankschemas. Die Klassen und ihre Struktur sind im wesentlichen aus den im OOS-Schema definierten Baustein- und Beziehungsarten hervorgegangen. Zusätzlich wurden die beiden virtuellen Schemaklassen „Entity“ und „Relation“ als Unterklassen der vordefinierten OPAL-Klasse „Object“ entworfen. Diese definieren allgemeine Eigenschaften von Bausteinen und Beziehungen.

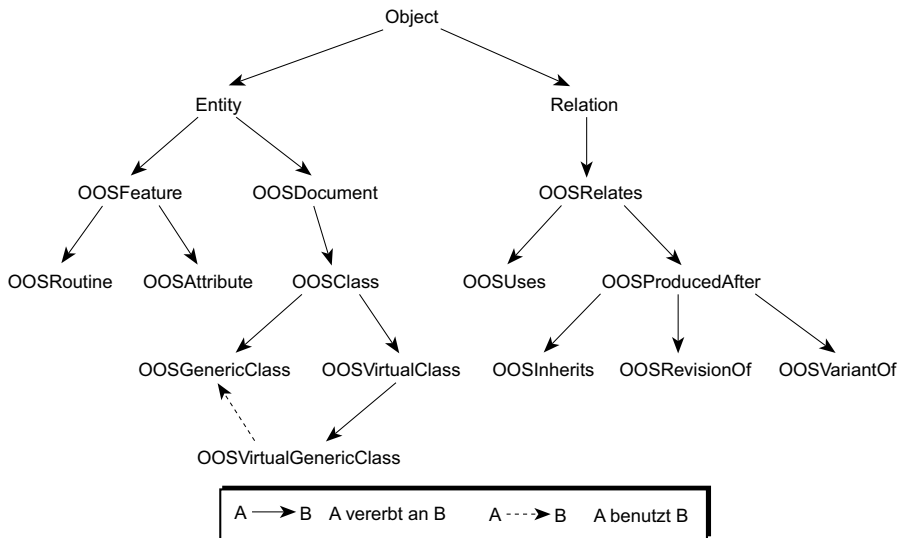


Abb. 7.6: Das Schema der ProWork-Datenbank

Auf die folgende Besonderheit sei speziell hingewiesen: Da GemStone nur die Einfachvererbung zwischen Klassen erlaubt, konnte die Schemaklasse „OOS-VirtualGenericClass“ nicht als Unterklasse von „OOSVirtualClass“ und „OOSGenericClass“ definiert werden, wie es konzeptionell sinnvoll gewesen wäre. Statt dessen ist sie als Unterklasse von „OOSVirtualClass“ im Schema definiert. Die Eigenschaften der Klasse „OOSGenericClass“ werden mithilfe einer benutzt-Beziehung zugänglich gemacht.

7.6 Die hybride Ausführung von ProST und Eiffel

Ein Architekturprototyp, der aus Komponenten besteht, die teils in ProST und teils Eiffel realisiert sind, kann nur dann bewertet werden, wenn eine technische Möglichkeit angeboten wird, um ihn auszuführen. Diese muß erlauben, daß aus Routinen, die in ProST formuliert sind, Eiffel-Routinen aufgerufen werden können. In diesem Abschnitt wird beschrieben, wie die hybride Ausführung von ProST und Eiffel konzipiert und realisiert ist.

7.6.1 Das gewählte Konzept

Hybride Programme sind Programme, die aus Bausteinen bestehen, die in unterschiedlichen Sprachen formuliert sind. In Bischofberger (1990) werden Werkzeuge klassifiziert, mit denen hybride Programme ausgeführt werden

können. Diese Klassifikation reicht vom Code-Generator, dessen Ergebnisse manuell mit vorhandenem Objekt-Code zusammengebunden werden müssen, bis hin zu einer Entwicklungsumgebung, in der verschiedene kooperierende Sprachinterpretierer ein hybrides Programm ausführen.

Für die hybride Ausführung von ProST und Eiffel wurde folgendes Konzept gewählt: ProST wird, da die Sprache in die Smalltalk-80 Programmierumgebung eingebettet ist, natürlich interpretiert. Die interpretative Entwicklungsumgebung wird um eine prozedurale Schnittstelle erweitert, mit der die Routinen der Klassen, die in Eiffel implementiert sind, aufgerufen werden können. Dazu muß der übersetzte Eiffel-Code zur Smalltalk-80 Programmierumgebung hinzugebunden werden.

7.6.2 Die Anbindung der Sprachen – Ein technischer Überblick

Die Laufzeitumgebung der Sprache Smalltalk-80, die die Basis von ProST ist, besteht im wesentlichen aus zwei Teilen: aus der *virtuellen Maschine* (VM) und aus dem sogenannten *Image*. Die virtuelle Maschine implementiert grundlegende Funktionen, damit das Smalltalk-80-System arbeiten kann. Das „Image“ speichert alle Objekte, die der Anwender erzeugt, den übersetzten Smalltalk-80-Code und die Werkzeuge der Programmierumgebung.

Smalltalk-80 in der Version 4.0 und höher erlaubt, sogenannte *User Defined Primitives* (UDP), die in C implementiert sind, zu verwenden. Wie dies technisch geschieht und was dabei zu beachten ist, wird eingehend in ParcPlace (1992) beschrieben. Mithilfe der UDP's kann die virtuelle Maschine um eigene C-Routinen erweitert werden. Diese Routinen können dann direkt über eine Smalltalk-80-Methode aus dem Image aufgerufen werden.

Die Möglichkeit, die virtuelle Maschine zu erweitern, sowie die Möglichkeit, Eiffel-Code in C-Bibliotheken übersetzen zu lassen, bilden die technische Basis, um Eiffel-Code an ProST-Code anzubinden. In Eiffel (1990) wird erläutert, wie Eiffel-Code in C-Bibliotheken übersetzt und mit C-Code zusammengebunden werden kann.

Auf dieser technischen Grundlage wurde eine Schnittstelle zwischen der virtuellen Maschine von Smalltalk-80 und dem in C übersetzten Eiffel-Code entwickelt (ProST-Eiffel-Schnittstelle). Abbildung 7.7 zeigt schematisch und ohne alle im Detail notwendigen Schritte, wie Eiffel-Code an ProST-Code angebunden wird.

Eiffel- und C-Compiler erzeugen aus dem Eiffel-Quell-Code ein Archiv, das den Objekt-Code enthält. Dieses wird zusammen mit dem Objekt-Code der initialen virtuellen Maschine und der ProST-Eiffel-Schnittstelle zu einer um den Eiffel-Code erweiterten, neuen virtuellen Maschine zusammengebunden. Mit dieser kann ein „Image“ geladen werden, in dem die Eiffel-Routinen aus ProST-Routinen aufgerufen werden können.

Die erweiterte virtuelle Maschine wird statisch zusammengebunden, ein dynamisches Anbinden von Eiffel-Code an die virtuelle Maschine ist nicht vorgesehen.

Um Eiffel-Objekte in ProST ansprechen und ihnen Nachrichten schicken zu können, existiert zu jedem Eiffel-Objekt ein korrespondierendes ProST-Objekt, das als dessen Stellvertreter angesehen werden kann. Es dient ausschließlich dazu, Nachrichten an sein Eiffel-Objekt weiterzuleiten.

Zu diesem Zweck

- muß das ProST-Objekt sein Eiffel-Objekt kennen,
- müssen die Rümpfe der ProST-Routinen verändert werden, damit die entsprechende Eiffel-Routine aufgerufen und ein eventuell produziertes Resultat zurückgeliefert werden kann.

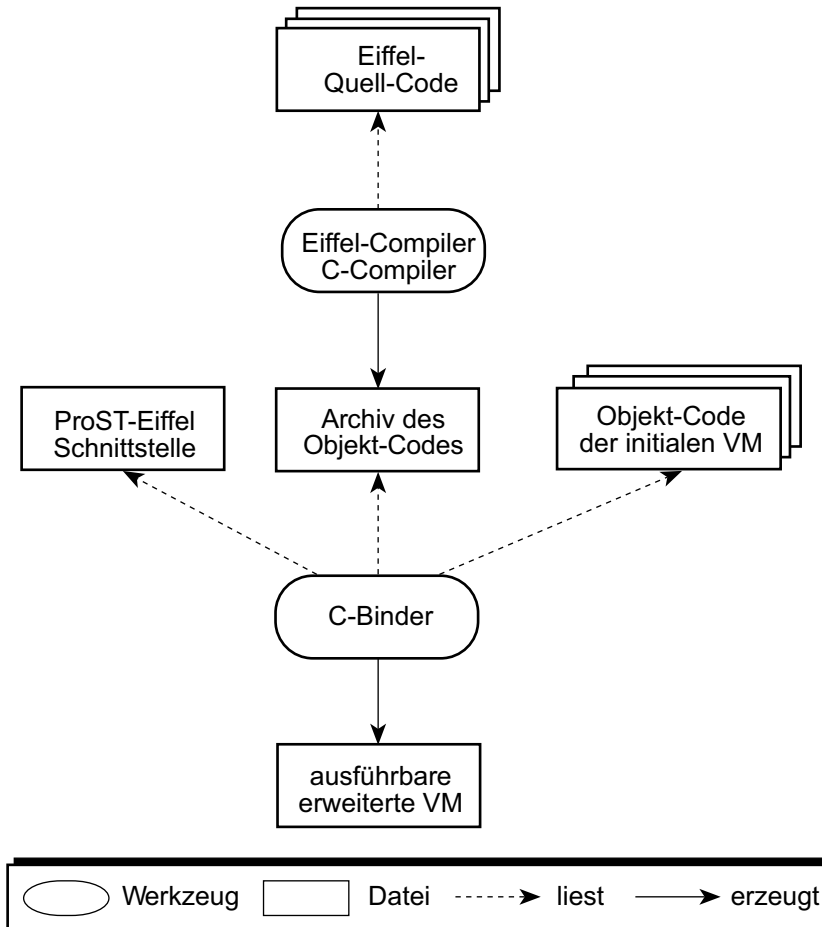


Abb. 7.7: Die Anbindung von Eiffel an ProST

Die Verbindung zwischen ProST- und Eiffel-Objekt wird dadurch hergestellt, daß jedes ProST-Objekt die Adresse des korrespondierenden Eiffel-Objekts speichert. Dies funktioniert jedoch nur dann, wenn sich die Speicheradresse des Eiffel-Objekts während seiner Lebenszeit nicht verändert.

Die notwendigen Modifikationen in den Rümpfen der ProST-Routinen werden automatisch vom Werkzeug ProST-Eiffel-I-Transformator vorgenommen. Dieser bestimmt aus der Anzahl der Parameter einer Routine sowie aus dem eventuellen Resultatstyp, welche Routine der ProST-Eiffel-Schnittstelle aufzurufen ist, und modifiziert den Rumpf entsprechend.

7.6.3 Die Realisierung der Schnittstelle

Um die Anbindung von Eiffel an ProST zu realisieren, mußten zwei Teile entwickelt werden: die in C implementierte ProST-Eiffel-Schnittstelle und der in ProST implementierte Zugriff auf diese Schnittstelle. Beide Teile werden nachfolgend kurz erläutert.

Die ProST-Eiffel-Schnittstelle

Sie arbeitet nach folgendem Konzept: Alle nach C übersetzten Eiffel-Routinen werden in einer Tabelle gespeichert. Soll einer der in der Tabelle gespeicherten Routinen aus ProST aufgerufen werden, so muß der ProST-Eiffel-Schnittstelle der Tabellenindex sowie eventuelle Parameter übergeben werden. Diese Informationen reichen aus, um die gewünschte Routine auszuwählen, sie mit Parameter zu versorgen und aufzurufen.

Der Zugriff auf die ProST-Eiffel-Schnittstelle

Die folgende Abbildung illustriert, wie eine Eiffel-Routine aus ProST aufgerufen wird.

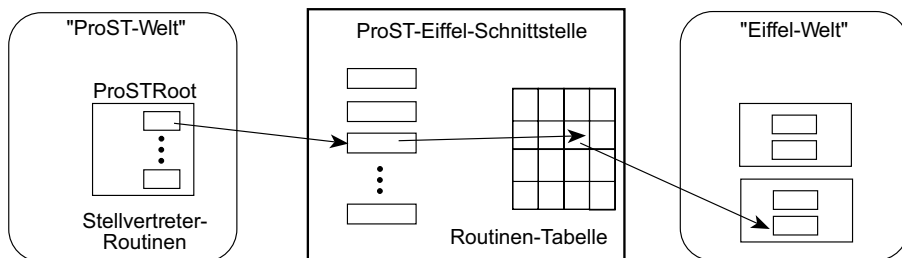


Abb. 7.8: Verbindung zwischen der ProST- und Eiffel-Welt

Damit die Routinen der ProST-Eiffel-Schnittstelle in ProST-Routinen verwendet werden können, werden sie als UDP's zur virtuellen Maschine hinzugebunden. In der Systemklasse „ProSTRoot“, die die Oberklasse aller ProST-Klassen ist, ist für jede der hinzugebundenen Routinen eine „Stellvertreter-Routine“ implementiert. Diese tun nichts anderes, als die dazu korrespondierende Routine der ProST-Eiffel-Schnittstelle aufzurufen. Die Stellvertreter-Routinen müssen immer dann verwendet werden, wenn eine Eiffel-Routine aus ProST aufgerufen werden soll.

Merkmale der Realisierung

Das Konzept, die ProST-Eiffel-Schnittstelle tabellengesteuert zu realisieren, führt auf der einen Seite dazu, daß die notwendige Tabelle immer neu erzeugt werden muß, wenn eine zusätzliche Eiffel-Klasse an ProST angebunden werden soll. Andererseits muß der Code der Schnittstelle, sowohl auf der C- als auch auf der ProST-Seite, nicht modifiziert werden, wenn neue Eiffel-Klassen angebunden werden sollen.

7.6.4 Die Werkzeugunterstützung

Um Eiffel-Code an ProST-Code anzubinden, stellt ProWork die Werkzeuge *ProST-Eiffel-Binder* und *ProST-Eiffel-I-Transformator* zur Verfügung. Nachfolgend wird der ProST-Eiffel-Binder beschrieben, der das zentrale Werkzeug ist, um die beiden Sprachen anzubinden. Anschließend wird gezeigt, welche Schritte notwendig sind, um den ProST-Code einer Klasse durch den entsprechenden Eiffel-Code zu ersetzen und welche Werkzeuge insgesamt daran beteiligt sind.

Der ProST-Eiffel-Binder

Als Eingabe benötigt das Werkzeug den Eiffel-Quell-Code aller Klassen, die zur virtuellen Maschine hinzugebunden werden sollen. Das Werkzeug arbeitet in folgenden Schritten:

- *Namenskonflikte auflösen*

Da die Eiffel-Routinen letztlich über C-Routinenbezeichner aus ProST angesprochen werden, muß gewährleistet werden, daß alle C-Routinenbezeichner paarweise verschieden sind. Zu Namenskonflikten kommt es aber immer dann, wenn zwei Klassen eine Routine mit gleichem Bezeichner deklarieren. Da die C-Routinenbezeichner der Eiffel-Klassen für den Anwender nicht sichtbar sind, kann eine Abbildungsvorschrift gewählt werden, die auf die Semantik der Bezeichner keine Rücksicht nimmt.

- *Objekt-Code-Archiv erstellen*
Der Eiffel-Quell-Code wird mithilfe des Eiffel- und C-Compilers entsprechend der gewählten Bezeichnerabbildung in ein Objekt-Code-Archiv übersetzt.
- *Routinentabelle erzeugen*
Anschließend wird die Routinentabelle für die ProST-Eiffel-Schnittstelle generiert und die Eingabedatei für den ProST-Eiffel-I-Generator erzeugt. Diese enthält alle Informationen, um auf der ProST-Seite die Anbindung der Eiffel-Klassen sicherzustellen.
- *Neue virtuelle Maschine erstellen*
Der Objekt-Code der ursprünglichen virtuellen Maschine, die ProST-Eiffel-Schnittstelle und das Archiv, das den Objekt-Code der Eiffel-Klassen enthält, werden zu einer ausführbaren neuen virtuellen Maschine zusammengebunden.

Die resultierende virtuelle Maschine enthält den Code, der in Eiffel implementierten Klassen. Dieser kann aus ProST-Routinen verwendet werden.

Die Zusammenarbeit der Werkzeuge

Eine Klasse, die prototypisch realisiert ist, wird folgendermaßen durch ihre Eiffel-Implementierung ersetzt:

- Zuerst wird mit dem ADL-Eiffel-Transformator aus der ADL-Formulierung der Klasse eine Eiffel-Beschreibung erzeugt.
- Nachdem die Klasse in Eiffel implementiert und getestet ist, wird der Eiffel-Code mit dem ProST-Eiffel-Binder zur neuen erweiterten virtuellen Maschine zusammengebunden.

Mit der erstellten virtuellen Maschine kann das „Image“ geladen werden, in dem der Architekturprototyp ausgeführt werden soll. Bevor dieser wieder gestartet werden kann, muß der Code der ProST-Routinen durch Aufrufe entsprechender Schnittstellenroutinen ersetzt werden. Dies leistet der ProST-Eiffel-I-Generator.

Abbildung 7.9 zeigt diese Schritte, die notwendigen Eingaben für die Werkzeuge und deren Produkte.

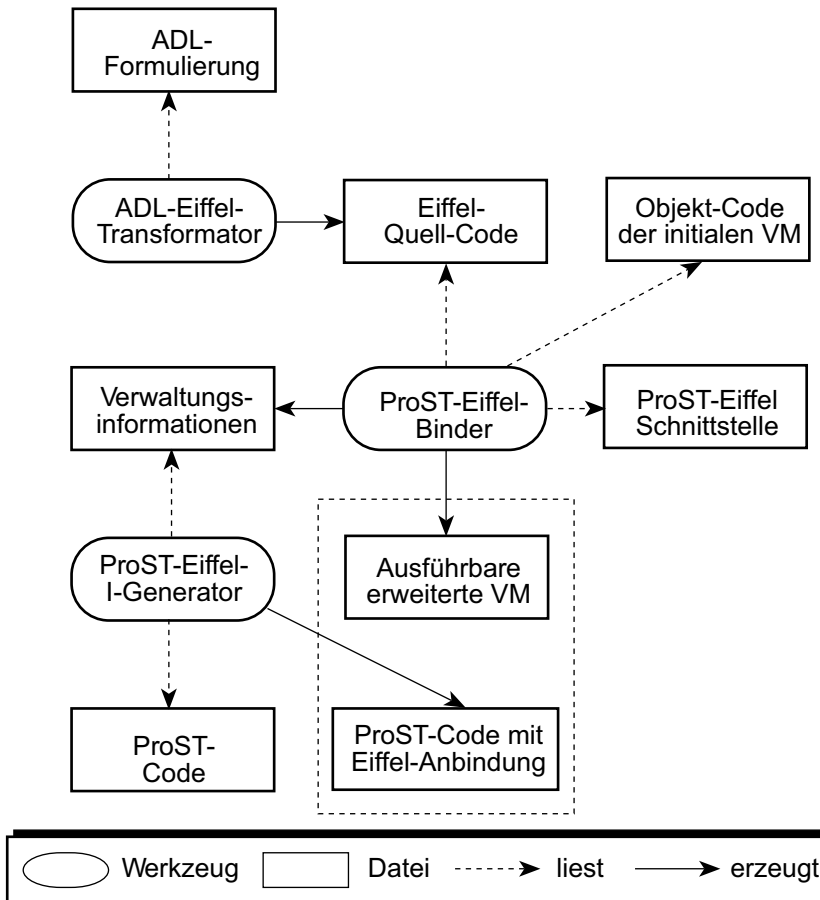


Abb. 7.9: Arbeitsschritte und Werkzeuge, um Eiffel an ProST anzubinden

7.6.5 Die Anbindung von Eiffel an ProST - Ein Beispiel

Abbildung 7.10 zeigt an einem einfachen aber im Detail ausgeführten Beispiel, wie eine ProST-Routine an eine entsprechende Eiffel-Routine angebunden wird. Als Beispiel wird die Routine „moveTo“ der Klasse „Rectangle“ verwendet.

- 1 Die Routine „moveTo“ hat einen Parameter und ist in der ProST-Klasse „Rectangle“ realisiert.
- 2 Die ProST-Routine wird durch den ProST-Eiffel-I-Generator so verändert, daß dort eine in der Systemklasse „ProSTRoot“ realisierte Eiffel-Schnittstellenroutine aufgerufen wird (p1s:class:par1). Als

Parameter erhält diese eine interne Kennung, die aus einem Klassen- und Routinenindex besteht und die Routine „moveTo“ eindeutig in der Routinentabelle identifiziert.

- Die in „ProSTRoot“ implementierte Schnittstellenroutine ruft lediglich eine Routine der ProST-Eiffel-Schnittstelle auf und versorgt diese mit den übergebenen Parametern. Diese ist als UDP an die virtuelle Maschine angebunden (<primitive: 10001>).

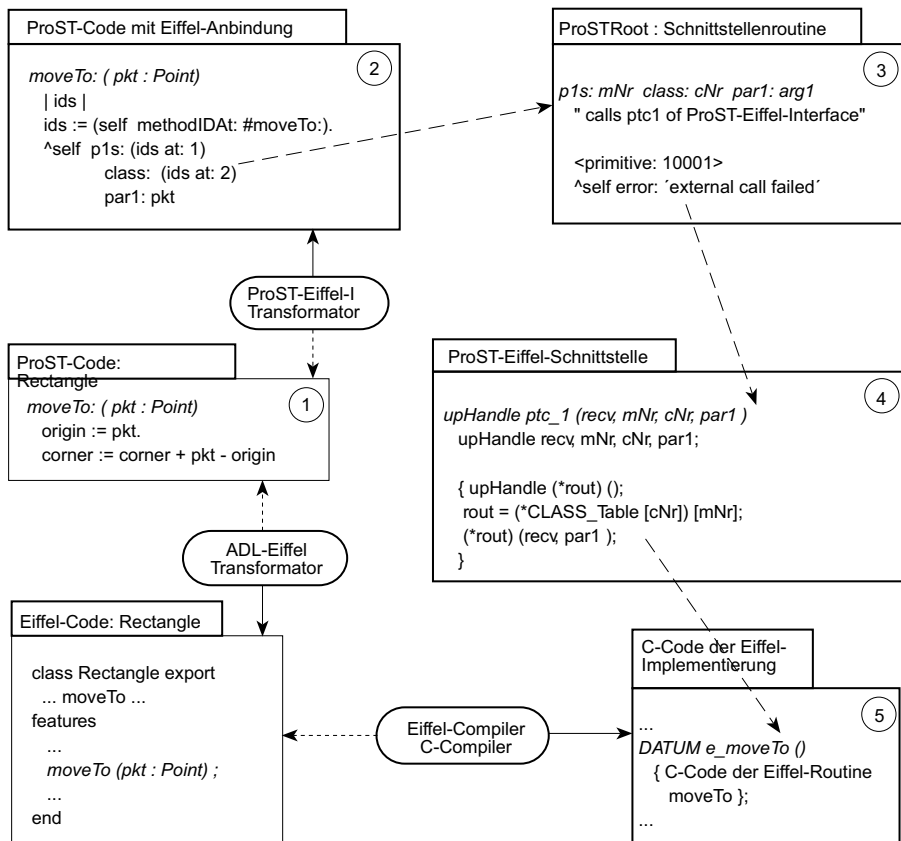


Abb. 7.10: Aufruf einer Eiffel-Routine aus ProST

- Aufgrund der übergebenen Identifikation und der vom ProST-Eiffel-Binder erzeugten Routinentabelle, in der die nach C übersetzten Eiffel-Routinen gespeichert sind, wird die C-Routine ausgewählt, die der Eiffel-Routine „moveTo“ entspricht.

5 Die ausgewählte C-Routine wird ausgeführt.

7.7 Sonstige Werkzeuge

ProWork stellt neben den bereits detailliert beschriebenen Werkzeugen einen Prototypkonfigurator, einen Datenbankmanager und ein integriertes Ausführungswerkzeug für Architekturprototypen zur Verfügung. Die Funktionalität und das Aussehen dieser Werkzeuge werden nachfolgend kurz skizziert.

7.7.1 Der Prototypkonfigurator

Mit einem Architekturprototyp sollen verschiedene Alternativen untersucht und bewertet werden. Dazu wird ein Werkzeug benötigt, um alternative Konfigurationen für den Architekturprototyp zu definieren. ProWork stellt zu diesem Zweck den *Prototypkonfigurator* zur Verfügung.

Der Prototypkonfigurator berücksichtigt, daß zu Klassen Revisionen und Varianten existieren können und daß sie entweder in ProST oder in Eiffel implementiert sind. Der Prototypkonfigurator stellt folgende Funktionen zur Verfügung:

- Es kann eine Standardkonfiguration erstellt werden. Dazu wählt das Werkzeug Klassen nach folgenden Gesichtspunkten aus:
 - Es wird immer die aktuelle Revision einer Klasse in die Konfiguration aufgenommen.
 - Existiert für eine Klasse eine Zielsprachenimplementierung in Eiffel, wird diese den prototypischen Implementierungen in ProST vorgezogen.

Die vom Werkzeug generierte Standardkonfiguration kann manuell angepaßt werden.

- Konfigurationen können in der ProWork-Datenbank gespeichert und wieder in das Werkzeug geladen werden.

Das Werkzeug ist in Abbildung 7.11 zu sehen. Folgende Informationen sind in der Darstellung eines Klassenbezeichners enthalten: Ist er fett geschrieben, ist die Klasse bereits in Eiffel implementiert. Ist er kursiv gesetzt, ist die Klasse noch prototypisch in ProST realisiert. Ansonsten existiert lediglich die ADL-Formulierung. Das Werkzeug ist aus fünf Fenstern aufgebaut, die folgende Inhalte zeigen.

- Im Fenster 1 werden alle Konfigurationen angezeigt, die bereits gebildet wurden und in der ProWork-Datenbank gespeichert sind. Diese können in das Werkzeug übernommen werden.
- Im Fenster 2 werden alle Klassen einer Architektur angezeigt, aus denen eine Konfiguration gebildet werden kann.

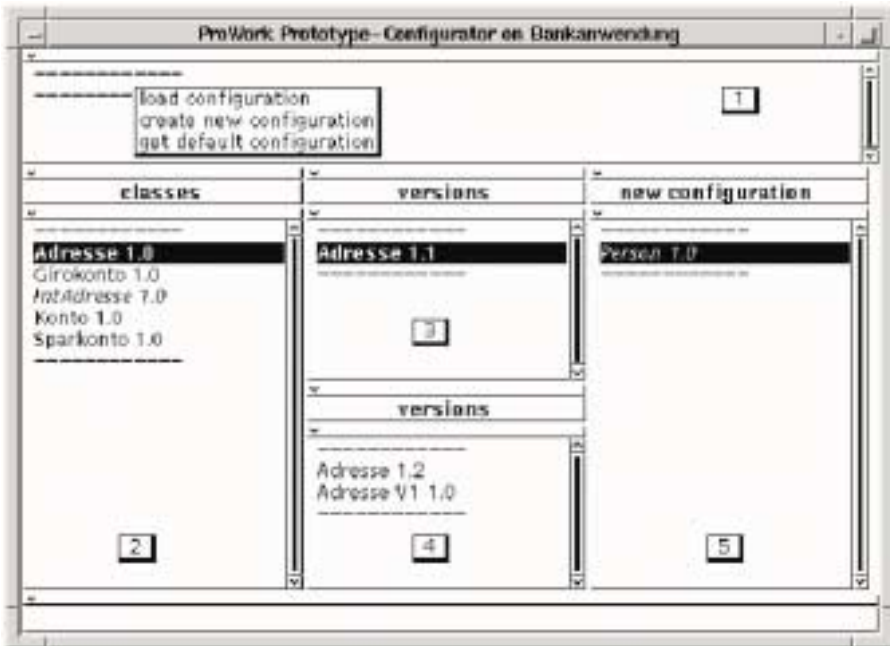


Abb. 7.11: Der Prototypkonfigurator

- Fenster 3 und Fenster 4 dienen dazu, um stufenweise den Baum, der durch die Versionen einer Klasse gebildet wird, zu betrachten. Dabei wird der Inhalt von Fenster 4 immer dann in Fenster 3 angezeigt, wenn der Baum der Versionen eine Stufe tiefer betrachtet werden soll. Das folgende Beispiel, das in Abbildung 7.11 zu sehen ist, soll die Arbeitsweise illustrieren:

Zur Klasse „Adresse“, die in Fenster 2 angezeigt wird, existieren Versionen. Wird sie selektiert, so werden die von ihr direkt abgeleiteten Versionen in Fenster 3 aufgelistet. Wird anschließend die Revision 1.1 im Fenster 3 ausgewählt, zu der es wiederum direkt abgeleitete Versionen gibt, so werden diese in Fenster 4 aufgelistet. Wird nun in Fenster 4 eine Version selektiert, zu der es weitere Versionen gibt, dann wird der Inhalt

von Fenster 4 in Fenster 3 dargestellt. Die Liste der hinzugekommenen Versionen ist in Fenster 4 zu sehen.

- Im Fenster 5 werden alle Klassen aufgelistet, die in die Konfiguration aufgenommen sind. Eine Klasse, die in die Konfiguration aufgenommen wird, wird nicht mehr in Fenster 2 angezeigt. Dadurch wird gewährleistet, daß jede Klasse nur einmal in einer Konfiguration enthalten ist.

7.7.2 Der Datenbankmanager

Alle Dokumente, die die Werkzeuge erzeugen, werden in einer ProWork-Datenbank gespeichert¹. ProWork stellt den *Datenbankmanager* zur Verfügung, um verschiedene projektspezifische Datenbanken zu erstellen und zu administrieren.

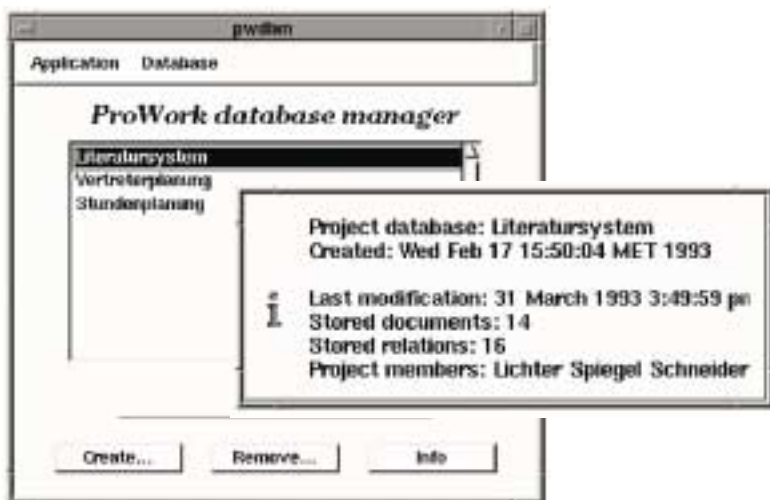


Abb. 7.12: Der Datenbankmanager

¹ Die Produkte des Eiffel-Compilers werden nicht in der Datenbank gespeichert, da dieser auf das UNIX-Dateisystem abgestimmt ist und eine eigene Dateistruktur erzeugt.

Mit dem Datenbankmanager können neue Datenbanken angelegt, Informationen zu existierenden Datenbanken abgefragt, und Datenbanken gelöscht werden.

Der Informationssatz, der zu einer Datenbank angezeigt wird, enthält folgende Angaben:

- Datum und Uhrzeit, wann die Datenbank erzeugt wurde.
- Datum und Uhrzeit der letzten Modifikation.
- Die Anzahl der gespeicherten Dokumente und Beziehungen.
- Die Namen der Mitarbeiter, die berechtigt sind, die Datenbank zu verändern.

Abbildung 7.12 ist ein Bildschirmabzug, der die Benutzungsschnittstelle des Werkzeugs zeigt. Im Fenster des Werkzeugs werden alle ProWork-Projekt-datenbanken aufgelistet, die aktuell vorhanden sind. Das Datenbank-Menü bietet Funktionen an, um durchgeführte Transaktionen zu bestätigen oder rückgängig zu machen. Die Funktionen „create“, „remove“ und „info“ können über Knöpfe aufgerufen werden.

7.7.3 Das Ausführungswerkzeug

Dieses Werkzeug faßt alle die Werkzeuge unter einer einheitlichen Benutzungs-schnittstelle zusammen, die immer dann verwendet werden, wenn ein Architekturprototyp gemäß einer mit dem Prototypkonfigurator erstellten Konfiguration ausgeführt werden soll.

Zu diesem Zweck kann zuerst der Eiffel-Compiler gestartet werden. Dieser erzeugt das C-Archiv der Klassen, die im Architekturprototyp in ihrer Eiffel-Implementierung ausgeführt werden sollen. Anschließend kann der ProST-Eiffel-Binder aufgerufen werden, der – nach den in Abschnitt 7.6 beschriebenen Schritten – die speziell auf die gewählte Konfiguration angepaßte virtuelle Maschine erzeugt.

Abbildung 7.13 zeigt das Aussehen des Werkzeugs. Im Textfenster des Werkzeugs werden die Ausgaben, die der Eiffel-Compiler und der ProST-Eiffel-Binder erzeugen, protokolliert. Die vom ProST-Eiffel-Binder erstellten virtuellen Maschinen, die an die Konfigurationen des Architekturprototyps angepaßt sind, können in Abhängigkeit vom ausgewählten Projekt in einem Listenfenster selektiert und anschließend gestartet werden.

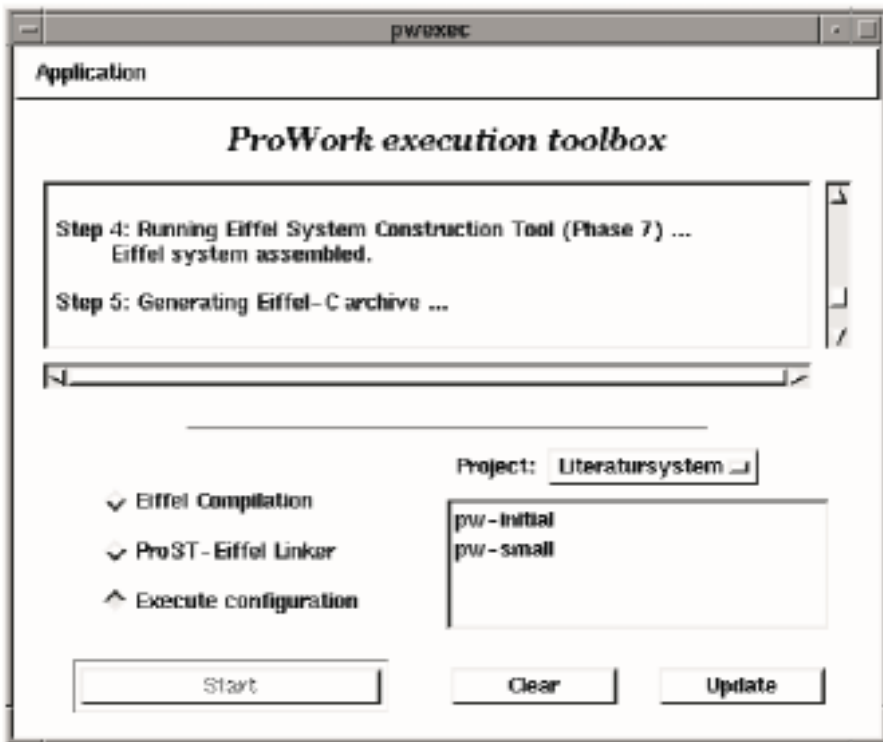


Abb. 7.13: Das Ausführungswerkzeug

Kapitel 8

Beispiel, Vergleich und Bewertung

In den vorangegangenen Kapiteln wird die Vorgehensweise PDSC vorgestellt, um Architekturprototypen schrittweise in das Zielprogramm zu überführen. Es wird ferner ein Software-Schema, die darauf angepaßten Sprachen sowie eine Werkstatt beschrieben, um Architekturprototypen für objektorientierte Anwendungen zu erstellen.

In diesem Kapitel wird diese Vorgehensweise an einem Beispiel illustriert. Anschließend wird der beschriebene Ansatz mit verwandten Arbeiten verglichen. Abschließend wird er bewertet, indem die Stärken den Schwächen und nicht gelösten Problemen gegenübergestellt werden.

8.1 Ein Beispiel

Im Rahmen dieses Textes ist es schwer, ein Beispiel zu präsentieren, das alle Gesichtspunkte ausführlich zeigt, die beim Software-Architektur-Prototyping nach der Vorgehensweise PDSC von Interesse sind. Insbesondere kann die Dynamik einer Prototypentwicklung nur unzureichend dargestellt werden.

Aus diesem Grund wird im anschließend präsentierten Beispiel nicht der vollständige Prozeß – von der Analyse bis zum fertigen Zielprogramm – gezeigt, sondern es werden speziell die folgenden Aspekte vorgestellt:

- die Formulierung des Architekturmodells in ADL
- die Entwicklung und Erprobung von Versionen
- der Übergang eines Programmbausteins von seiner Prototyp- in seine Zielsprachenimplementierung

8.1.1 Die Problemstellung

Die Problemstellung, die in diesem Text vorgestellt und bearbeitet werden soll, darf nicht zu kompliziert sein, damit auf eine umfangreiche Erläuterung und Analyse verzichtet werden kann. Sie muß allerdings erlauben, daß die Aspekte, die im Zusammenhang mit dem Architektur-Prototyping von Interesse sind, illustriert werden können.

Aus diesen Gründen wurde als Beispiel die Entwicklung eines Bibliographie-Systems gewählt. Diese Aufgabenstellung und ein dazu realisiertes System wird in Budde et al. (1987b) vorgestellt. In Budde et al. (1987c) wird diese Aufgabenstellung verwendet, um objektorientiertes Entwerfen vorzuführen. Die Autoren präsentieren eine Analyse der Problemstellung, entwerfen eine software-technische Lösung und setzen diese in eine Smalltalk-80 Implementierung um.

Die Aufgabenstellung und der dazu erstellte Entwurf werden für das Beispiel verkürzt und so abgeändert, daß die im vorigen Abschnitt gewählten Aspekte gezeigt werden können. Die vollständige Aufgabenstellung ist in der ersten zitierten Arbeit auf Seite 60 zu finden. Sie wird nachfolgend nur soweit wiedergegeben, wie für das Verständnis des Beispiels notwendig ist.

Ein Bibliographie-System enthält Einträge, die zu Literaturverzeichnissen zusammengefaßt werden können. Jeder Eintrag identifiziert einen Text. Die Einträge können sich auf unterschiedliche Textarten beziehen. Dies sind Bücher, Artikel, Berichte usw. Die Einträge enthalten artspezifische Angaben, um den Text zu identifizieren. Es sollen die folgenden bibliographischen Einträge vorgesehen werden:

Bücher

Referenzkürzel			
Autor(en)	Titel		
Verlag	Ort	Jahr	
Stichwörter	Kommentare		

Technische Berichte

Referenzkürzel			
Autor(en)	Titel	Reportangabe	
Institution	Ort	Jahr	
Stichwörter	Kommentare		

Artikel in einem Buch

Referenzkürzel		
Autor(en)	Titel	Seiten
Herausgeber	Buchtitel	
Verlag	Ort	Jahr
Stichwörter	Kommentare	

Artikel in einer Zeitschrift

Referenzkürzel		
Autor(en)	Titel	Seiten
Zeitschrift	Jahrgang	Nummer
Verlag	Ort	Jahr
Stichwörter	Kommentare	

Das Referenzkürzel wird automatisch vom Bibliographie-System aus den Angaben zu den Autoren und dem Jahr der Veröffentlichung erzeugt. Zu jedem Eintrag läßt sich eine Referenz erstellen, wie sie im Literaturteil von Veröffentlichungen üblich ist.

8.1.2 Das Architekturmodell

Um das Architekturmodell darzustellen, werden Bildschirmabzüge des ADL-Architecteditors gezeigt. Die Graustufe, mit der das Symbol einer Klasse hinterlegt ist, hat folgende Bedeutung:

- *weiß*: Die Klasse ist in ADL formuliert.
- *hellgrau*: Die Klasse ist in ProST implementiert. Der ProST-Code wird im Architekturprototyp verwendet.
- *dunkelgrau*: Die Klasse wird augenblicklich in Eiffel codiert. Im Architekturprototyp wird die ProST-Implementierung verwendet.

dunkelgrau mit starker Umrahmung: Die Klasse ist in Eiffel implementiert. Der Eiffel-Code wird im Architekturprototyp verwendet.

Das Architekturmodell in Abbildung 8.1 zeigt nur einen Teil der Klassen, die für die software-technische Lösung der Aufgabenstellung notwendig sind. Alle Klassen liegen in ihrer ADL-Formulierung vor. Es ist eine Vererbungshierarchie von Klassen zu sehen, die die bibliographischen Einträge realisieren. Diese Hierarchie ist dadurch entstanden, daß die Daten, die die einzelnen Einträge charakterisieren, auf Gemeinsamkeiten untersucht wurden. Gemeinsame Daten wurden in Oberklassen zusammengefaßt.

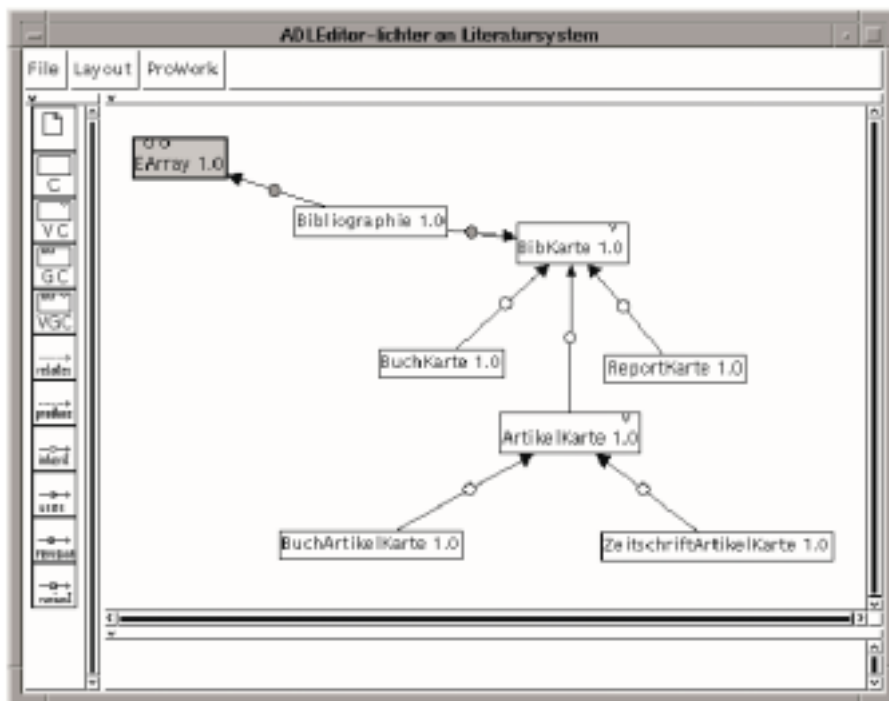


Abb. 8.1: Architekturmodell des Bibliographie-Systems (Szene 1)

An der Spitze dieser Hierarchie steht die virtuelle Klasse „BibKarte“. Sie definiert alle Attribute, die bei allen Arten von Einträgen vorhanden sind, sowie Routinen, um diese Attribute zu modifizieren und um Referenzen zu erzeugen. Die Klasse „BibKarte“ ist virtuell, weil ein Objekt davon kein sinnvoller bibliographischer Eintrag ist und weil sie die virtuelle Routine „generateReference“ definiert. Diese kann nicht realisiert werden, weil dazu Attribute verwendet werden, die erst in konkreten Unterklassen eingeführt werden. Die Klasse „BibKarte“ legt fest, daß alle ihre konkreten Unterklassen diese Routine redefinieren müssen.

Nachfolgend ist ein Auszug aus der textuellen ADL-Formulierung der Klasse „BibKarte“ abgebildet.

attributes

authors -> Array[String] is private
"Liste der Autoren."
title -> String is public
"Titel der Veroeffentlichung."
year -> Integer is public
"Erscheinungsjahr"
city -> String is public
"Ort der Veroeffentlichung."
comment -> String is public
"Kommentar zur Veroeffentlichung."
keywords -> Array[String] is private
"Stichwoerter"

routines

initialize is public
"Initialisiert die Attribute."
virtual generateReference -> String is public
"Erzeugt die vollst. Referenz."
shortReference -> String is public
"Erzeugt das Referenzkuerzel."
addKeyword: (k : String) is public
"Fuegt ein Stichwort hinzu."
removeKeyword: (k : String) -> Boolean is public
"Loescht ein Stichwort."
keywordsAsString -> String is public
"Liefert eine Zeichenkette aller Stichwoerter".
addAuthor: (a : String) is public
"Fuegt einen Autor hinzu."
removeAuthor: (a : String) -> Boolean is public
"Loescht einen Autor."
authorsAsString -> String is public
"Liefert eine Zeichenkette aller Autoren."
putTitle: (t : String) is public
"Titel der Veroeffentlichung zuordnen."
...

Die Klasse „Bibliographie“ ist so modelliert, daß sie in einem Behälter die einzelnen Einträge aufnimmt. Sie definiert das Attribut „label“, das eine Bibliographie näher bezeichnet. Mit den Routinen der Klasse können Operationen auf einer Bibliographie ausgeführt werden, beispielsweise Suchen

nach Referenzen oder Einfügen eines neuen Eintrags. Die ADL-Formulierung der Klasse in einer ersten einfachen Version sieht folgendermaßen aus:

attributes

cardContainer -> Array[BibKarte] is private
 "Speichert die Eintraege."
 label -> String is public
 "Etikett der Bibliographie."

routines

initialize is public
 "Initialisiert die Attribute."
 addCard: (c : BibKarte) is public
 "Fuegt einen Eintrag in die Bibliographie ein."
 removeCard: (ref : String) -> Boolean is public
 "Loescht den Eintrag mit dem Referenzkuerzel ref."
 searchCards: (qu : String) -> Array[BibKarte] is public
 "Liefert alle Eintraege, die der Anfrage qu genügen."
 ...

8.1.2 Erprobung von Versionen

Abbildung 8.2 zeigt eine weitere Szene im Prozeß der Architekturmodellierung. Das Architekturmodell wurde in zwei Punkten weiterentwickelt.

a) Neue Revision der Klasse „BibKarte“

Für die Klasse „BibKarte“ wurde eine neue Revision erzeugt (BibKarte 1.1). Diese wurde gebildet, weil bei der Prüfung der Klasse erkannt wurde, daß Autoren nicht als eine Zeichenkette betrachtet werden dürfen. Autoren sollen als eine eigenständige Klasse in der Architektur modelliert werden.

Dies hat den Vorteil, daß Informationen zu Autoren lokal in einer Klasse definiert und über Routinen erfragt und modifiziert werden können. Als Konsequenz dieser Überlegung wurde die Klasse „Autor“ in das Architekturmodell aufgenommen. Sie definiert die Attribute Vorname(n), Nachname und Arbeitsgebiet sowie entsprechende Routinen. Die ADL-Formulierung der Klasse „Autor“ wird nicht gezeigt, da sie denkbar einfach ist.

Die neue Revision der Klasse „BibKarte“ wurde aus einer Arbeitskopie des Dokuments „BibKarte 1.0“ erzeugt. Nachdem alle notwendigen Verände-

rungen an der Arbeitskopie durchgeführt waren, wurde sie zur aktuellen Revision der Klasse „BibKarte“. Die Beziehungen, die zwischen dem initialen Dokument und anderen Dokumenten bestanden, wurden entsprechend nachgezogen. Dies unterstützt der ADL-Architektureditor.

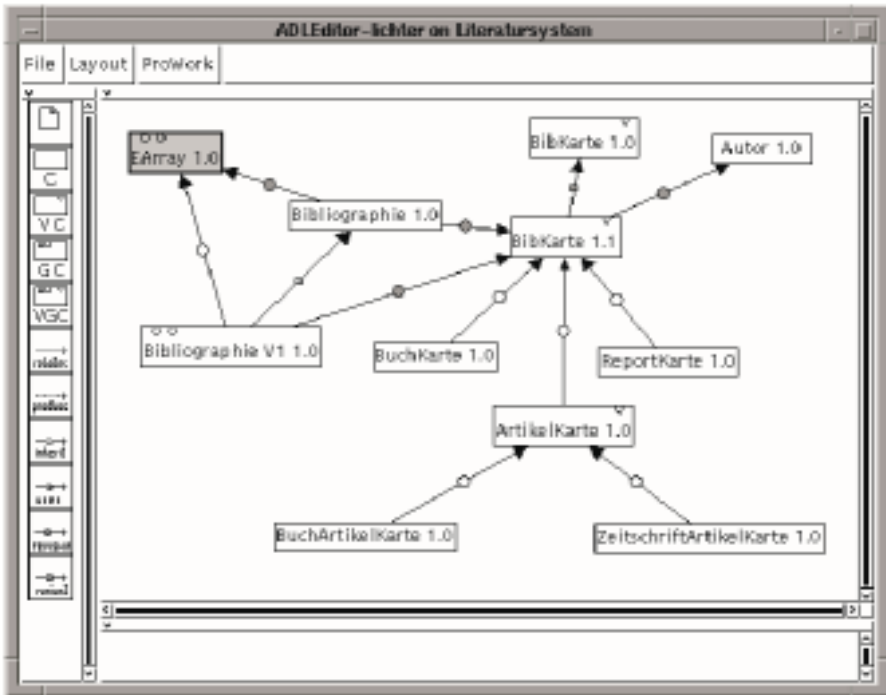


Abb. 8.2: Architekturmodell des Bibliographie-Systems (Szene 2)

Die Revision 1.1 der Klasse „BibKarte“ verwendet die Klasse „Autor“ (benutzt-Symbol zwischen den beiden Klassen). Dazu mußte ihre ADL-Formulierung modifiziert werden. Diese sieht in der Revision 1.1 nun folgendermaßen aus.

attributes

```
authors -> Array[Autor] is private  
"Liste der Autoren."
```

```
...
```

routines

```
addAuthor: (a : Autor) is public  
"Fuegt einen Autor hinzu."  
removeAuthor: (a : Autor) -> Boolean is public  
"Loescht einen Autor."
```

```
...
```

Die Attribute und Routinen, die unverändert aus der Revision 1.0 übernommen wurden, werden aus Platzgründen nicht noch einmal gezeigt.

b) Zusätzliche Variante der Klasse „Bibliographie“

Zur Klasse „Bibliographie“ wurde eine Variante entwickelt (Bibliographie V1 1.0). Diese unterscheidet sich von der ursprünglichen Version im wesentlichen dadurch, daß sie als Unterklasse zur Klasse „Array“ definiert wurde¹. Bei der neuen Variante wird eine Bibliographie – entsprechend der Spezialisierungssemantik der Vererbung – als ein spezielles „Array“ angesehen. In der ursprünglichen Version wurde dagegen ein „Array“ benutzt, um eine Bibliographie zu realisieren.

¹ Die Klasse „Array“ ist bereits in der Eiffel-Bibliothek vorhanden. Damit sie in der ProST-Umgebung verwendet werden kann, wurde die Klasse „EArray“ als Stellvertreter in die ProST-Bibliothek aufgenommen. Alle Stellvertreterklassen sind mit einem vorangestellten „E“ markiert, damit es keine Namenskonflikte mit vorhandenen Smalltalk-80-Klassen gibt.

```
generic Card -> BibKarte
```

```
attributes
```

```
label -> String is public  
  "Etikett der Bibliographie"
```

```
routines
```

```
initialize is public  
  "Initialisiert die Attribute."  
addCard: (c : Card) is public  
  "Fuegt eine Karte in die Bibliographie ein."  
removeCard: (ref : String) -> Boolean is public  
  "Loescht die Karte mit dem Referenzkuerzel ref."  
searchCards: (qu : String) -> Array[Card] is public  
  "Liefert alle Karten, die der Anfrage qu genuegen."  
...  
...
```

Diese Entwurfsentscheidung verändert die neue Variante der Klasse „Bibliographie“ in folgenden Punkten:

- Die Klasse ist generisch. Sie definiert einen formalen generischen Parameter, der die Elemente einschränkt, die eine Bibliographie aufnehmen kann. Es sollen nur Objekte der Klasse „BibKarte“ und deren Unterklassen in einer Bibliographie verwaltet werden.
- Das in der ursprünglichen Version definierte Attribut „cardContainer“ wird nicht mehr benötigt.

Nachdem die Klassen „Autor“, „BibKarte“, „BuchKarte“ und die beiden Versionen der Klasse „Bibliographie“ prototypisch in ProST implementiert sind, können die Alternativen der Architektur untersucht werden. Dazu können zwei Konfigurationen des Architekturprototyps gebildet werden. Diese bestehen aus folgenden Dokumenten.

Konfiguration 1: Autor 1.0, BibKarte 1.1, BuchKarte 1.0, Karteikasten 1.0

Konfiguration 2: Autor 1.0, BibKarte 1.1, BuchKarte 1.0, EArray 1.0, Karteikasten V1 1.0,

8.1.3 Der Übergang zum Zielprogramm

Abschließend wird anhand des Beispiels gezeigt, wie die ProST-Formulierung einer Klasse im Architekturprototyp durch eine Eiffel-Implementierung ersetzt wird. Dabei wird von dem in der folgenden Abbildung gezeigten Zustand der Modellierung ausgegangen:

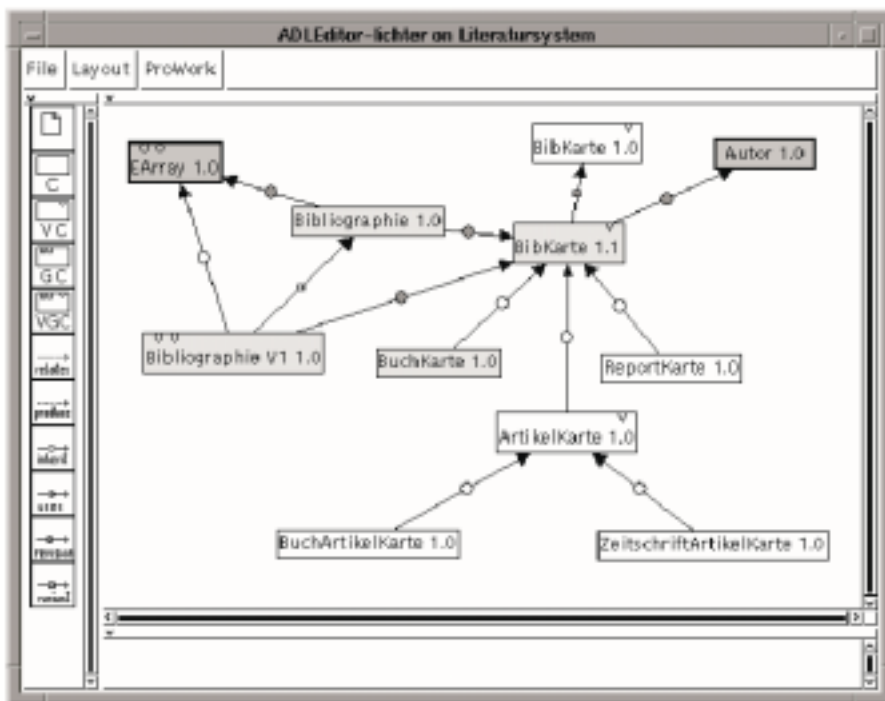


Abb. 8.3: Architekturmodell des Bibliographie-Systems (Szene 3)

Die Klasse „Autor“ ist bereits in Eiffel implementiert und wird auch in ihrer Eiffel-Implementierung im Architekturprototyp verwendet. Als nächstes soll die Revision 1.1 der Klasse „BibKarte“ in eine Eiffel-Implementierung überführt werden. Dies ist möglich, weil die Klasse „Autor“, die benutzt wird, bereits in Eiffel realisiert ist. Nachdem ein Eiffel-Rahmen aus der ADL-Formulierung der Klasse „BibKarte“ erzeugt wurde, können die definierten Routinen implementiert werden.

Ein Ausschnitt des Eiffel-Codes der Klasse „BibKarte“ ist nachfolgend abgebildet:

```
deferred class BibKarte
export
  authors, title, year, city, comment, keywords,
  initialize, shortReference ... generateReference

feature
  authors : Array [ Autor ] ;
  ...
  authorCount : Integer;
  ...
  shortReference : String is
  local
    short, lastname : String;
    i : Integer;
  do
    short := "[";
    from i := authors.lower until i = authorCount
    loop
      lastname := authors.item(i).lastname;
      short.extend (lastname.item (1) );
      short.extend (lastname.item (2) );
    end ;
    short.append(year.out);
    short.append("]");
    result := short;
  end; -- shortReference
  ...
  generateReference : String is
  deferred
  end; -- generateReference
end -- BibKarte
```

Der Eiffel-Code der Klasse „BibKarte“ wird anschließend, wie in Kapitel 7 beschrieben wird, mit dem ProST-Eiffel-Binder an die in ProST formulierten Klassen gebunden. Dadurch wird er im Architekturprototyp anstelle des ProST-Codes ausgeführt.

Der Prozeß, in dem Architekturalternativen erprobt, stabile Bausteine in Eiffel implementiert und anschließend in den Architekturprototyp integriert werden, dauert solange an, bis alle Bausteine in einer Eiffel-

Implementierung vorliegen. Der Architekturprototyp wird dadurch schrittweise zum Zielprogramm.

8.2 Vergleich mit verwandten Arbeiten

In diesem Abschnitt werden verwandte Ansätze und Projekte beschrieben. Diese werden kurz charakterisiert und mit dem in dieser Arbeit vorgestellten Ansatz verglichen.

Es wurden Projekte ausgewählt, die methodische und technische Ergebnisse für das Prototyping geliefert haben und zu denen Unterlagen verfügbar sind. Im einzelnen werden die Projekte TOPOS, PROTOS, CAPS und ProLab vorgestellt. Bei allen Projekten wurden spezielle Prototyp-Entwicklungs-umgebungen erstellt.

8.2.1 TOPOS

Überblick

Das Projekt TOPOS (**TO**ol set for **Pr**ototyping **O**riented **S**oftware development) wurde an der Universität Zürich begonnen und später an der Universität Linz weitergeführt. Die Ziele, die im Projekt TOPOS erreicht werden sollten, sind in Pomberger (1987) formuliert. Zusammengefaßt sollten

- theoretische und technische Grundlagen für das Prototyping entwickelt und
- eine Werkzeugumgebung realisiert werden, die das Prototyping unterstützt.

Die realisierte Prototyp-Entwicklungs-umgebung TOPOS ist in Abbildung 8.4 zu sehen. Sie besteht aus drei Werkzeuggruppen:

- Werkzeuge zum Benutzungsschnittstellen-Prototyping
- Werkzeuge zum Architektur-Prototyping
- Werkzeuge zum Projekt-Management

Zwei Werkzeuggruppen werden zur Verfügung gestellt, um Prototypen für Benutzungsschnittstellen zu erstellen: UICT (**U**ser **I**nterface **C**onstruction **T**ool) und DICE (**D**ynamic user **I**nterface **C**reation **E**nvironment). UICT bietet Werkzeuge, um Benutzungsschnittstellen entweder grafisch oder textuell zu erstellen und interpretieren zu lassen. Der Aufbau einer Benutzungsschnittstelle wird in der Sprache UISL (**U**ser **I**nterface **S**pecification **L**anguage) formuliert, die speziell für diesen Zweck entwickelt

wurde. Die UISL-Spezifikation einer Benutzungsschnittstelle kann mit Code-Generatoren nach Modula-2 oder C übersetzt werden. Die Werkzeuge von UICT werden detailliert in Keller (1988) beschrieben.

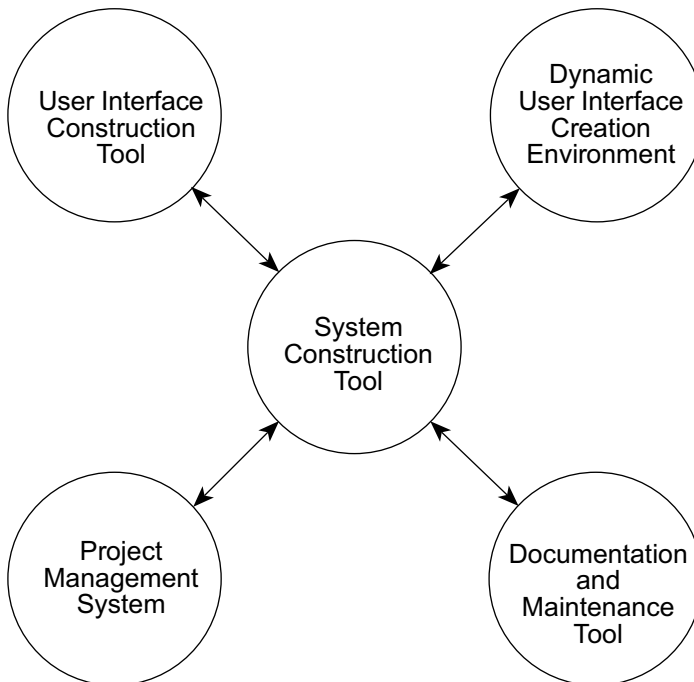


Abb. 8.4: Aufbau von TOPOS - übernommen aus Plösch (1993)

DICE stellt, ähnlich wie UICT, Werkzeuge bereit, um Prototypen von Benutzungsschnittstellen zu konstruieren. DICE generiert im Gegensatz zu UICT jedoch C++ Klassen, die an das application framework ET++ angepaßt sind. DICE wird in Pree (1991) vorgestellt.

Das Herz von TOPOS ist das Werkzeug SCT (**S**ystem **C**onstruction **T**ool). Es wird in Bischofberger (1990) beschrieben und erlaubt, daß Prototypen ausgeführt werden können, die aus Teilen bestehen, die in verschiedenen Sprachen erstellt sind und auf verschiedene Arten ausgeführt werden. Dieses wird als "hybrides Ausführen eines hybriden Programms" bezeichnet. Mit SCT können Teile eines Prototyps

- interpretiert werden,
- simuliert werden

- oder der übersetzte Objekt-Code wird ausgeführt.

Dazu enthält SCT einen Modula-2-Interpreter und einen Simulator. Mit SCT können einzelne Programmbausteine und deren Zusammenspiel im Prototyp untersucht werden, bevor sie vollständig implementiert sind.

Das TOPOS-Projekt-Managementsystem erlaubt es, Dokumente, die zu einem Projekt gehören, zu verwalten. Es stellt Werkzeuge zur Verfügung, um Dokumente zu betrachten, zu modifizieren und um Versionen der Dokumente zu verwalten. Diese Werkzeuge werden in Schmidt (1989) vorgestellt.

Die Ergebnisse des Projekts TOPOS werden in Bischofberger (1992) zusammengefaßt.

Abgrenzung

Der Schwerpunkt beim Projekt TOPOS liegt darauf, Werkzeuge zu konzipieren und zu entwickeln, die Prototyping unterstützen. Ein Software-Schema, nach dem Architekturen modelliert werden, existiert nicht oder ist nur implizit durch die Möglichkeiten definiert, die die Sprachen Modula-2 und C zur Verfügung stellen, um ein Programm zu modularisieren. Die Art und Weise, wie mit den TOPOS-Werkzeugen Prototypen erstellt werden, kann als „outside-in“ oder „von der Benutzungsschnittstelle getrieben“ bezeichnet werden. Dies unterscheidet TOPOS vom Architektur-Prototyping, wie es in dieser Arbeit vorgestellt wird.

8.2.2 PROTOS

Überblick

Hallmann (1988) beschreibt einen Ansatz, mit dem Anforderungen an transaktionsorientierte Systeme formuliert und daraus bereits während der Anforderungsanalyse Prototypen entwickelt werden können. Dazu wurde die Methode REMOS und die Sprache RELOS entwickelt, sowie die Entwicklungsumgebung PROTOS realisiert. Diese Komponenten sind im Sinne des Systemdreiecks aufeinander abgestimmt.

Die Methode REMOS (**RE**quirements **M**ethodology for transaction-oriented **O**pen **S**ystems) wurde entwickelt, um systematisch Anforderungen für transaktionsorientierte Systeme zu formulieren. Die Methode ist formal definiert und geht davon aus, daß transaktionsorientierte Systeme nur inkrementell erstellt werden können. REMOS ist eine Anleitung, um

Prototypen inkrementell auf der Basis von sukzessiv definierten Transaktionen zu erstellen. Transaktionen werden in Szenarien definiert. Szenarien beschreiben Abläufe und werden in der Sprache RELOS (**RE**quirements **L**anguage for transaction-oriented **O**pen **S**ystems) formuliert. RELOS ist applikativ, ausführbar und speziell für die transaktionsorientierte Anforderungsdefinition von verteilten transaktionsorientierten Systemen geeignet.

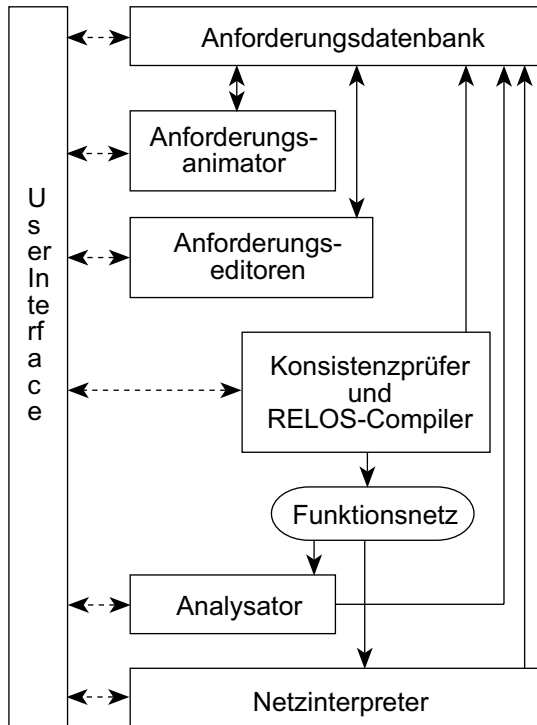


Abb. 8.5: Die Werkzeuge von PROTOS - (vereinfacht, nach Hallmann, 1988)

PROTOS (**PRO**Totyping environment for transaction-oriented **O**pen **S**ystems) stellt Werkzeuge zur Verfügung, um Anforderungen für transaktionsorientierte Systeme in der Sprache RELOS einzugeben und um anschließend einen Prototyp auszuführen.

Zu diesem Zweck wird die RELOS-Formulierung mit dem RELOS-Compiler in ein Funktionsnetz übersetzt, das interpretiert werden kann. Der Netzinterpretier erstellt eine Protokolldatei, die von einem Animator visualisiert wird. Dadurch kann der Benutzer das von ihm in Transaktionen spezifizierte Verhalten eines Systems am Bildschirm in grafisch animierter

Art verfolgen. PROTOS ist so konzipiert, daß alle Werkzeuge auf einer sogenannten Anforderungsdatenbank arbeiten.

Abgrenzung

REMOS und die Werkzeuge in PROTOS sind für eine spezielle Klasse von Anwendungssoftware konzipiert. Die Prototypen werden erstellt, um Anforderungen zu ermitteln und zu validieren, nicht um Architekturen zu bewerten. REMOS definiert eine Standardarchitektur für transaktionsorientierte Systeme und Subsysteme, die nicht modifiziert werden kann.

8.2.3 CAPS

Überblick

CAPS (**C**omputer **A**ided **P**rototyping **S**ystem) wurde an der Naval Postgraduate School in Monterey entwickelt, um schnell und einfach Prototypen für komplexe Realzeit-Anwendung erstellen zu können.

Dazu wurde die Prototyping-Sprache PSDL (**P**rototype **S**ystem **D**escription **L**anguage) entwickelt. Diese Sprache ist geeignet, um Anforderungen für Realzeit-Anwendungen zu formulieren. Sie basiert auf einem Berechnungsmodell, in dem Operatoren, die über Datenströme miteinander kommunizieren, eine Berechnung durchführen. Mit PSDL können dementsprechend Operatoren und Datenströme definiert werden. Die Sprache ist weiterhin so konzipiert, daß PSDL-Spezifikationen einfach mit den Möglichkeiten der Programmiersprache Ada implementiert werden können. PSDL wird in Luqi (1988a) vorgestellt.

Die Werkzeuge der CAPS-Entwicklungsumgebung sind in Abbildung 8.6 dargestellt. Es werden grafische und syntax-orientierte Editoren angeboten, um eine PSDL-Spezifikation zu erstellen. Diese Spezifikationen werden in normalisierter Form einem „Design-Manager“ übergeben. Dieser hat die Aufgabe, Spezifikationen, wiederverwendbare Bausteine sowie Versionen und Alternativen der Spezifikationen mithilfe einer Datenbank zu verwalten. Um eine PSDL-Spezifikation auszuführen, werden ein Übersetzer, ein statischer und ein dynamischer „Scheduler“ bereitgestellt. Diese erzeugen ausführbaren Ada-Code. Die Werkzeuge von CAPS werden im Überblick in Luqi (1988b) beschrieben. In Luqi (1990) wird gezeigt, wie eine PSDL-Spezifikation in Ada-Code transformiert wird.

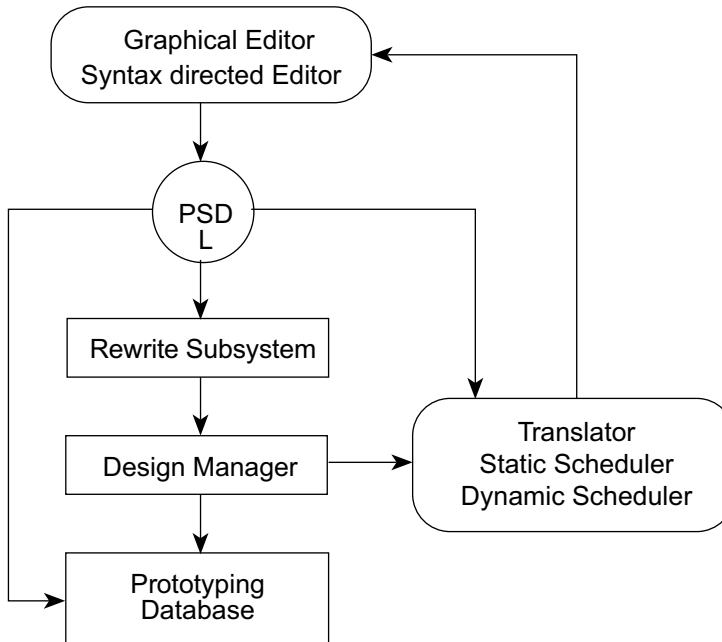


Abb. 8.6: Die Werkzeuge von CAPS - entsprechend Luqi (1988b)

Abgrenzung

CAPS ist, wie schon das System PROTOS, auf eine spezielle Klasse von Anwendungen zugeschnitten. In CAPS sind Prototypen ausführbare Spezifikationen, mit denen die funktionalen Anforderungen für Realzeitanwendungen validiert werden können.

Der Übergang zum Zielsystem wird von CAPS unterstützt. Dazu kann die PSDL-Spezifikation erweitert werden, so daß sie zusätzlich zur Prototypbeschreibung auch den Ada-Code der einzelnen Operatoren enthält.

8.2.4 ProLab

ProLab ist ein Akronym, das nach seinen Entwicklern sowohl für **Prototyping-Labor** als auch für **Prolog-Labor** steht. Es wurde am Institut für Systemtechnik der Gesellschaft für Mathematik und Datenverarbeitung in Bonn entwickelt und wird in Bäcker et al. (1988a) und Bäcker et al. (1988b) beschrieben.

In ProLab wurde versucht, die Sprache Prolog für verschiedene Zwecke einzusetzen: als Entwurfs-, Prototyping- und Zielsprache. Dabei wurde davon ausgegangen, daß Prototyp und Zielsystem in ihrer Architektur übereinstimmen und daß der Prototyp und das Zielsystem in Prolog implementiert werden. Die Architekturbeschreibung der Systeme wird in der Prolog-Datenbasis gespeichert und kann mit Prolog-Regeln bearbeitet werden.

ProLab bietet Werkzeuge, die eine evolutionäre Systementwicklung unterstützen, bei der Prototypen erstellt werden. Diese Vorgehensweise wird in Budde et al. (1986) beschrieben.

Zu den Werkzeugen von ProLab gehören unter anderem:

- Der *Environment Manager*: Mit ihm kann der Benutzer ProLab an seine persönlichen Bedürfnisse anpassen.
- Das *Session Logbook*: Mit ihm können Kommandosequenzen aufgezeichnet und wiederholt werden.
- Das *Data Dictionary*: Es speichert und verwaltet Relationen zwischen den sogenannten Entwicklungsgegenständen (Dateien und Prozeduren) und Entwicklungszuständen (Dokumenten).
- Der *Test Driver*: Er erlaubt, daß gespeicherte Tests durchgeführt werden.

Die Werkzeuge sind so miteinander vernetzt, daß sie den Wechsel zwischen Editieren, Laden, Ausführen und Testen, der für das Prototyping typisch ist, erleichtern und unterstützen.

Abgrenzung

In ProLab werden Entwurf, Prototyp und Zielprogramm in Prolog formuliert. Ein Übergang vom Prototyp zum Zielsystem findet dementsprechend nicht explizit statt und muß auch nicht unterstützt werden.

Prolog ist aufgrund der kompakten Schreibweise, den vordefinierten mächtigen Datenstrukturen und der interpretativen Ausführung geeignet, um Prototypen zu erstellen. Dies wird bereits in Venken (1984) festgestellt. Es muß jedoch bezweifelt werden, ob Prolog auch als Entwurfs- und Zielsprache verwendet werden sollte.

8.3 Zusammenfassung

Bevor die Ergebnisse dieser Arbeit bewertet werden, soll der „Rote Faden“, der sich durch die gesamte Arbeit zieht, noch einmal deutlich gemacht werden.

In dieser Arbeit wird ein Ansatz für das Prototyping präsentiert, bei dem die Software-Architektur mit einem Prototyp - dem Architekturprototyp - untersucht wird.

Dazu wurde zuerst herausgearbeitet, wie Prototyping mit der Terminologie der allgemeinen Modelltheorie beschrieben werden kann. Ein Prototyp ist gemäß dieser Theorie ein spezielles Computermodell. Er besitzt alle Merkmale, die ein Modell auszeichnen. Diese Überlegungen führten zu vier Grundfragen, die gestellt und beantwortet werden müssen, wenn ein Prototyp entwickelt werden soll.

Auf der Basis dieser Vorüberlegungen wurde die Modellierung der Software untersucht. Es wurden drei Abstraktionsebenen definiert, die für die Software-Modellierung relevant sind: die Schema-, die Modell- und die Ausprägungsebene. Da Software-Entwicklung Dokumentenentwicklung ist, sind Dokumente die Elemente, aus denen ein Software-Modell aufgebaut werden kann. Es wurden allgemeine Beziehungsarten erörtert, mit denen modelliert werden kann, wie die Dokumente im Prozeß der Software-Entwicklung entstehen und wie sich die einzelnen Dokumente im Sinne von Versionen weiterentwickeln. Die vorgestellten Beziehungsarten wurden in einem allgemeinen Software-Schema - dem DS-Schema - formuliert.

Im nächsten Schritt wurde ein allgemeiner Architekturbegriff für Dokumente eingeführt. Eine Dokumentenarchitektur faßt - nach der gegebenen Definition - eine Menge von Dokumenten zusammen, die ein definiertes Kriterium erfüllen. Dieser Architekturbegriff wurde anschließend auf die Architektur der Software übertragen. In der Software-Architektur sind aus der gesamten Dokumentenmenge nur diejenigen Dokumente enthalten, die für den Einsatz, das Verständnis und die Wartung des Zielprogramms notwendig sind. Die Dokumente, die einzig für den Konstruktionsprozeß der Software benötigt wurden, scheiden aus.

Die Überlegungen zur Software-Modellierung und Architekturbildung führten zu einer Klassifikation von Software-Modellierungsansätzen. Diese Klassifikation definiert die Software-Architekturmodellierung als einen speziellen Ansatz der Software-Modellierung. Sie führt weiterhin das

Software-Architektur-Prototyping als Spezialisierung der Software-Architekturmodellierung ein. Charakteristisch für das Software-Architektur-Prototyping ist, daß dabei ein Modell der Programmarchitektur entsteht, das ausgeführt werden kann.

Auf der Basis dieser konzeptionellen Ergebnisse wurde anschließend eine Vorgehensweise für das Software-Architektur-Prototyping vorgestellt, die PDSC genannt wird. Sie sieht vor, daß der konstruierte Architekturprototyp schrittweise in das Zielprogramm überführt wird. Es wurde diskutiert, warum dies sinnvoll ist und welche Vorteile diese Vorgehensweise hat. Weiterhin wurde beschrieben, welche Aktivitäten bei PDSC durchgeführt werden müssen, wie der Übergang im einzelnen vonstatten geht, welche Sprachen dazu notwendig sind und welche Werkzeuge diese Vorgehensweise operational unterstützen.

Das DS-Schema wurde daraufhin an die speziellen Erfordernisse der Vorgehensweise PDSC angepaßt. Das resultierende Software-Schema ist das PDSC-Schema. Mit diesem Schema kann Software gemäß PDSC modelliert werden; es ist jedoch noch nicht an eine spezielle Entwurfsmethode angepaßt.

Im letzten Teil der Arbeit wurde gezeigt, wie die vorgestellten Ideen zum Software-Architektur-Prototyping konstruktiv umgesetzt werden können. Dies wurde am Beispiel der objektorientierten Software-Entwicklung getan. Dazu wurde zuerst das OOS-Schema erstellt, das aus dem PDSC-Schema hervorgegangen ist. Dieses Schema ist geeignet, um objektorientierte Programmarchitekturen zu modellieren. Es wurden Sprachen vorgestellt, um Architekturmodelle gemäß diesem Schema zu erstellen, entsprechende Architekturprototypen zu realisieren und das Zielprogramm zu implementieren. Als Zielsprache wurde Eiffel gewählt; die Software-Modellierungssprache ADL und die Prototyping-Sprache ProST basieren weitgehend auf Eiffel und Smalltalk-80.

Abschließend wurden in dieser Arbeit Werkzeuge präsentiert, die das Prototyping objektorientierter Architekturen gemäß PDSC unterstützen. Diese Werkzeuge sind in der Werkstatt ProWork zusammengefaßt.

8.4 Bewertung des Ansatzes

Software-Architektur-Prototyping, die Vorgehensweise PDSC sowie die vorgestellten Sprachen und Werkzeuge können nur im praktischen Einsatz bewertet werden. Dies war im Rahmen dieser Arbeit nicht möglich.

Inwieweit das Software-Architektur-Prototyping für den praktischen Einsatz geeignet ist, kann dementsprechend auch nicht festgestellt werden.

Unabhängig davon können aus der Sicht des Autors die Stärken dieses Ansatzes formuliert und den noch nicht gelösten Problemen gegenübergestellt werden. Dies wird nachfolgend getan.

8.4.1 Stärken des Ansatzes

Der vorgestellte Ansatz hat folgende Stärken und Vorteile:

- Es wird ein Begriffsnetz für das Software-Architektur-Prototyping eingeführt, das einerseits aus der Terminologie der allgemeinen Modelltheorie und andererseits aus Begriffsdefinitionen zum Prototyping hervorgegangen ist. Dabei wird insbesondere geklärt, wie Architektur-Prototyping, Architekturmodellierung und Software-Modellierung zusammenhängen.
- Der vorgestellte dokumentenbasierte Ansatz zur Software-Modellierung integriert, auch wenn über die einzelnen Modellelemente diskutiert werden kann, mehrere Aspekte der Software-Entwicklung: das Entstehen der Dokumente im Konstruktionsprozeß, die Entwicklung der einzelnen Dokumente in Versionen und die semantische Beziehungen zwischen den einzelnen Dokumenten.
- Die beiden allgemeinen Software-Schemata, das DS- und das PDSC-Schema, sind offen und flexibel entwickelt. Sie bieten Ansatzpunkte, um an spezielle Erfordernisse angepaßt zu werden. Dazu zählen beispielsweise unternehmensinterne Entwicklungs-Standards und konkrete Entwurfsmethoden. Das OOS-Schema zeigt beispielhaft, wie das PDSC-Schema erweitert werden kann.
- Software-Architektur-Prototyping führt letztlich dazu, daß bereits frühzeitig die Architektur der Software zum Arbeitsschwerpunkt wird. Dies ist wichtig, da die entworfenen Strukturen besonders die Qualität der resultierenden Software beeinflussen. Es ist dabei vorteilhaft, wenn – wie in dieser Arbeit vorgeschlagen – ein Formalismus verwendet wird, um die entstehenden Strukturen zu beschreiben. Dadurch wird es möglich, schon frühzeitig die entworfene Architektur auf Konsistenz zu prüfen und zu bewerten.

8.4.2 Schwächen und nicht gelöste Probleme

Zu folgenden Punkten werden in dieser Arbeit keine Lösungen angeboten:

- Es werden nur wenige Aussagen gemacht, wie das Software-Architektur-Prototyping und die Vorgehensweise PDSC organisatorisch in konkreten Projekten umgesetzt werden kann. Dies ist jedoch notwendig, wenn Projekte, die arbeitsteilig organisiert sind, nach dieser Vorgehensweise durchgeführt werden sollen.
- Die Entwicklungsgeschichte einzelner Dokumente im Konstruktionsprozeß wurde als ein wichtiges Thema angesehen. Es wurden jedoch nur soweit Lösungsansätze formuliert, wie für die Modellierung der Software im Rahmen des Architektur-Prototyping notwendig gewesen sind. Eine detaillierte Betrachtung dieses Themas konnte und sollte nicht durchgeführt werden. Dieselbe Aussage gilt für Verwaltung von Konfigurationen.

Es ist jedoch denkbar, daß die präsentierten Ideen zur Software-Modellierung als Basis verwendet werden können, um neue Ansätze für die Verwaltung und Konfigurierung von Dokumenten zu entwickeln.

- Es werden keine Aussagen gemacht, wie existierende Dokumente und Programmbausteine in den Modellierungsprozeß und den Prozeß der Prototypkonstruktion einfließen können. Dieser Aspekt muß konzeptionell untersucht und technisch umgesetzt werden, da in der Regel Systeme oder Teilsysteme vorhanden sind, die in neuen Produkten verwendet werden sollen. Völlig neuartige Projekte sind dagegen eher der Ausnahmefall.
- Die entwickelten Software-Schemata (DS-, PDSC- und OOS-Schema) bieten keine Möglichkeit, die modellierten Dokumente im Sinn einer „ist-Teil-von-Hierarchie“ auf einer höheren Abstraktionsstufe zusammenzufassen. Dies ist jedoch notwendig, wenn die Anzahl der Dokumente sehr groß wird. Entsprechende Modellelemente könnten beispielsweise „Ordner“ oder „Subsysteme“ sein.

Es sei in diesem Zusammenhang darauf hingewiesen, daß es Ansätze gibt, um objektorientierte Programmbausteine zu gruppieren. Gruppierungen von Klassen werden als Klassenbibliotheken, Klassenkategorien oder auch „Cluster“ bezeichnet. Alle Gruppierungsarten sind jedoch immer reine Behälter, die keine Beziehungen zueinander haben. Es existiert zur Zeit noch kein Konzept, um Klassen zu gruppieren und Klassengruppen zu strukturieren, das die Semantik der „erbt-von-

Beziehung“ und der „benutzt-Beziehung“ zwischen Klassen berücksichtigt.

Der Ansatz zum Software-Architektur-Prototyping, bei dem der Architekturprototyp schrittweise in das Zielprogramm überführt wird, ist – so wie er in dieser Arbeit präsentiert wird – lediglich ein Kern für eine praxisgeeignete Vorgehensweise. Die geschilderten Probleme müssen bearbeitet und praktikable Lösungen dazu formuliert werden, um den vorgestellten Prototyping-Ansatz abzurunden. Nur so kann er für den praktischen Einsatz geeignet sein.

Literatur

- Ahmed, S., A. Wong, D. Sriram, R. Logcher (1992): Object-oriented database management systems for engineering: A comparison, **Journal of Object-Oriented Programming**, Vol. 5, No. 3, pp 27 - 44.
- Bäcker, A., R. Budde, K. Kuhlenkamp, A. Meckenstock, K.-H. Sylla, H. Züllighoven (1988a): **ProLab - A Prolog Programming Environment - User's Manual**, GMD-F2G2, St. Augustin.
- Bäcker, A., R. Budde, N. Dierichsweiler, K. Kuhlenkamp, A. Meckenstock, K.-H. Sylla, H. Züllighoven (1988b): Eine Übersicht über die Prolog-Programmierungsumgebung ProLab, in G. Snelting (Hrsg.) **Sprachspezifische Programmierungsumgebungen**, Workshop der Fachgruppe „Implementierung von Programmiersprachen“, pp 141 - 155.
- Baker, P.L. (1990): Object-Oriented Programming in Ada Using the Phylum Prespecifications, **Journal of Object-Oriented Programming**, January/February, pp 21-25.
- Bischofberger, W. (1990): Prototyping-Oriented Incremental Software Development - Paradigms, Methods, Tools and Implications, **Dissertation, Johannes Kepler Universität Linz**.
- Bischofberger, W., R. Keller (1989): Enhancing the Software Life Cycle by Prototyping, **Structured Programming**, Vol. 10, No. 1, pp 47-59.
- Bischofberger, W., G. Pomberger (1992): **Prototyping-Oriented Software Development - Concepts and Tools**, Springer-Verlag.
- Böcker, H.-D., M. Pawlitschek (1992): VICK: a visualization construction kit, **Journal of Object-Oriented Programming**, Vol. 4, No. 8, pp, 8-14.
- Boehm, B.W. (1976): Software Engineering. **IEEE Transactions on Computers**, Vol. 25, No. 12, pp 1226-1241. Boehm, B.W. (1988): A Spiral model of Software Development and Enhancement. **IEEE Computer**, May 1988, pp 61-72.

- Boehm, B.W., T.E. Gray, T. Seewaldt (1984): Prototyping versus Specification: A Multiproject Experiment. **IEEE Transactions on Software Engineering**, Vol. SE-10, No. 3, pp 290-302.
- Booch, G. (1991): **Object-Oriented Design with Applications**, Benjamin/Cummings Publishing Company, Redwood City.
- Budde, R., K. Kuhlenkamp, K.-H. Sylla, H. Züllighoven (1986): Prototypenbau bei der Systemkonstruktion, **Angewandte Informatik**, 4 und 5.
- Budde, R., M.-L. Christ-Neumann, K.-H. Sylla (1992a): Tools and Materials, an Analysis and Design Metaphor, G. Heeg, B. Magnusson and B. Meyer (eds.), **Proceedings of the Seventh International Conference TOOLS Europe '92**, Prentice Hall, pp 135-146.
- Budde, R., M.-L. Christ-Neumann, K.-H. Sylla, H. Züllighoven (1992b): Erfahrungen beim objektorientierten Entwerfen und Analysieren, **Arbeitspapiere der GMD**, Nr. 653.
- Budde, R., K. Kautz, K. Kuhlenkamp, H. Züllighoven (1992c): **Prototyping An Approach to Evolutionary System Development**. Springer-Verlag, Berlin.
- Budde, R., K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (eds) (1984): **Approaches to Prototyping**. Springer Verlag, Berlin.
- Budde, R., K. Kuhlenkamp, K.H. Sylla (1987a): Konzepte des Prototyping. In **Requirements Engineering '87**, GMD-Studien Nr. 121, Gesellschaft für Mathematik und Datenverarbeitung mbH, Sankt Augustin, pp 75-94.
- Budde, R., K. Kuhlenkamp, K.H. Sylla, H. Züllighoven (1987b): Bib - ein Bibliographie-System, H.-J. Hoffmann (Hrsg.) **Smalltalk verstehen und anwenden**, Hanser Verlag, pp 59 -88.
- Budde, R., K. Kuhlenkamp, K.H. Sylla, H. Züllighoven (1987c): Programm-entwicklung mit Smalltalk, H.-J. Hoffmann (Hrsg.) **Smalltalk verstehen und anwenden**, Hanser Verlag, pp 89 -121.
- Champeaux de, D., P. Faure (1992): A comparative study of object-oriented analysis methods, **Journal of Object-Oriented Programming**, March/April, pp 21-33.

- Coad, P., E. Yourdon (1991): **Object-Oriented Analysis**, second edition, Yourdon Press, Englewood Cliffs.
- Coad, P., E. Yourdon (1992): **Object-Oriented Design**, Yourdon Press, Englewood Cliffs.
- Constantine, L.L., E. Yourdon (1979): **Structured Design: Fundamentals of a discipline of computer programs and systems design**, Prentice-Hall, Englewood Cliffs.
- DeRemer, F., H.H. Kron (1976): Programming in the large versus programming in the small, **IEEE Transactions on Software Engineering**, Vol. SE-2, No. 2, pp 80-86.
- Diederich, L.G., J. Milton (1987): Experimental Prototyping in Smalltalk-80, **IEEE Software**, May, pp 50-64.
- Doberkat, E., D. Fox (1989): **Software Prototyping in SETL**, Teubner Verlag, Stuttgart.
- Duncan, A.C. (1982): Prototyping in Ada: A Case Study. **ACM SIGSOFT Software Engineering Notes**, Vol. 7, No. 5, pp 50-53.
- Eiffel (1990): **Eiffel: The Environment**, Interactive Software Engineering Inc, TR-EI-5/UM.
- Fairley, R. (1985): **Software Engineering Concepts**, McGraw-Hill Book Company, New York.
- Floyd, C. (1984): A Systematic Look at Prototyping. In Budde, R., K. Kuhlenskamp, L. Mathiassen, H. Züllighoven (Hrsg.) **Approaches to Prototyping**, Springer-Verlag, Berlin.
- Frühau, K., J. Ludewig, H. Sandmayr (1988): **Software-Projektmanagement und - Qualitätssicherung**, vdf, Zürich.
- Gamma, E., A. Weinand, R. Marty (1988): Integration of a Programming Environment into ET++ A Case Study, **Proceedings of the European Conference on Object-Oriented Programming ECOOP '89**, Cambridge University Press, pp 283-287.

- Glinz M., H.J. Huser, J. Ludewig (1985): SEED - A database system for software engineering environments. Blaser, Pistor (Hrsg.) **Datenbanksysteme für Büro, Technik und Wissenschaft**, Informatik-FB 94, Springer, pp 121-126.
- Goldberg, A. (1984): **Smalltalk-80 The Interactive Programming Environment**, Addison-Wesley Publishing Company.
- Goldberg, A., D. Robson (1983): **Smalltalk-80 The Language and its Implementation**. Addison-Wesley Publishing Company.
- Gordon, S., J. Bieman (1991): Rapid Prototyping and Software Quality: Lessons From Industry. **Technical Report CS-91-113, Colorado State University**, Department of Computer Science.
- Hallmann, M. (1985): Klassifizierung von Prototypingansätzen in der Softwareentwicklung. **Forschungsbericht des Fachbereichs Informatik der Universität Dortmund**, Nr. 209.
- Hallmann, M. (1988): Eine transaktionsorientierte operationale Methode zur Anforderungserfassung für das Prototyping. **Dissertation, Bericht Nr. 276/88, Abteilung Informatik der Universität Dortmund**.
- Heuer, A. (1992): **Objektorientierte Datenbanken**, Addison-Wesley Publishing Company.
- IEEE (1990): **IEEE Standard Glossary of Software Engineering Terminology**, IEEE Std 610.12-1990, The IEEE Inc., 345 East 27th Street, N.J. 10017-2394.
- Jackson, M. (1983): **System Design**, Prentice Hall International, Englewood Cliffs.
- Keller, R. (1988): **Prototyping-orientierte Systemspezifikation**, Dr. Kovac Verlag, Hamburg.
- Kieback, A., H. Lichter, M. Schneider-Hufschmidt, H. Züllighoven, (1992): Prototyping in industriellen Software-Projekten - Erfahrungen und Analysen, **Informatik-Spektrum**, Vol. 15, No. 2, Springer-Verlag, pp 65-67.
- Kanath, Y., J. Smith (1992): Experiences in C++ and object-oriented Design, **Journal of Object-Oriented Programming**, Vol. 5, No.7, pp 23-28.

- Krasner, G., S. Pope (1988): A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, **Journal of Object-Oriented Programming**, Vol. 1, No. 3, pp 26-48. Lanz, K. (1986): **The Prototyping Methodology**, Prentice-Hall.
- Lehman, M.M. (1980): Programs, Life Cycles, and Laws of Software Evolution. **IEEE Transactions on Software Engineering**, Vol. 68, No. 9, September 1980, pp 1060-1076.
- Lewis, J.A., S.M. Henry, D.G. Kafura, R.S. Schulman (1992): On the relationship between the object-oriented paradigm and software reuse: an empirical investigation, **Journal of Object-Oriented Programming**, July/August, pp 35-41. Lichter, H., K. Schneider (1993): vis-Avis - An Object-Oriented Framework for Graphical Design Tools. In Rix, E., E.G. Schlechtendahl (eds.) Proceedings of the IFIP-Workshop "**Interfaces in Industrial Systems for Production and Engineering**", March 1993, Darmstadt, Elsevier North Holland (to be published).
- Lientz, B.P., E.B. Swanson, G.E. Tompkins (1978): Characteristics of Application Software Maintenance. **Communications of the ACM**, Vol. 21, No. 6, pp 466-471.
- Liskov, B., S. Zilles (1974): Programming with abstract data types. **ACM Sigplan Notices**, Vol. 9, No. 4, pp 50-59.
- Ludewig J., H. Färberböck, H. Lichter, H. Matheis, E. Wallmüller (1987): Software-Entwicklung durch schrittweise Komplettierung. in Requirements Engineering '87, **GMD-Studien Nr 121**, Gesellschaft für Mathematik und Datenverarbeitung mbH, Sankt Augustin, pp 113-124.
- Ludewig, J. (1982): Computer aided specification of process control software, **IEEE COMPUTER**, Vol. 15, No. 5, pp 12-20.
- Ludewig, J. (1988): **Einführung in die Informatik**, vdf Zürich.
- Ludewig, J. (1989): Modelle der Software-Entwicklung: Abbilder oder Vorbilder? **Softwaretechnik Trends**, Band 9, Heft 3, pp 1-12.

- Ludewig, J., M. Glinz, H. Huser, G. Matheis, H. Matheis, M.F. Schmidt (1985a): Spades-A Specification and Design System and its Graphical Interface, **Proceedings of the 8th. Int. Conference on Software Engineering**, London, pp 83-89.
- Ludewig, J., M. Glinz, H. Matheis (1985b): Software-Spezifikation durch halbformale anschauliche Modelle, in H.R. Hansen (Hrsg.) GI/OCG/ÖGI- Jahrestagung 1985, **Informatik-Fachbericht 108**, Springer-Verlag, pp193-204.
- Luft, A. (1984): Zur Bedeutung von Modellen und Modellierungs-Schritten in der Softwaretechnik, **Angewandte Informatik**, AI-5/84, pp 189-196.
- Luqi (1990): Automated prototyping and data translation. **Data & Knowledge Engineering**, Vol. 5, No. 2, pp 167-177.
- Luqi, V. Berzins, R. Yeh (1988a): A Prototyping Language for Real-Time Software. **IEEE Transactions on Software Engineering**, Vol. 14, No. 10., pp 1409-1423.
- Luqi, M. Ketabchi (1988b): A computer aided prototyping system. **IEEE Software**, Vol. 5, No. 2, pp 66-72.
- Matheis, H. (1990): Informationsverwaltung für Software-Projekte, **Dissertation, Universität Stuttgart**, Institut für Informatik.
- Mattern, F., H. Mehl (1989): Diskrete Simulation - Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung, **Informatik-Spektrum**, Nr 12, pp 198-210.
- Meyer, B. (1988): **Object-Oriented Software Construction**. Prentice Hall, Series in Computer Science.
- Meyer, B. (1992): **Eiffel the Language**, Prentice Hall, Object-Oriented Series.
- Nagl, M. (1990): **Softwaretechnik: Methodisches Programmieren im Großen**. Springer Verlag, Berlin.
- ParcPlace (1992): **ObjectWorks\Smalltalk - User's Guide**, ParcPlace Systems Inc.
- Parnas, D.L. (1972): On the criteria to be used in decomposing systems into modules. **Communications of the ACM**, Vol. 15, No. 12, pp 1053-1058.

- Patton, B. (1983): Prototyping - a nomenclature problem. **ACM SIGSOFT Software Engineering Notes**, Vol. 8, No. 2, pp 14-16.
- Plösch, R., H. Rumersdorfer, C. Weinreich (1993): TOPOS: A Prototyping-Oriented Open CASE System, **Proceedings of Requirements Engineering '93**, Teubner, Stuttgart.
- Pomberger, G., W. Bischofberger, R. Keller, D. Schmidt (1987): Prototyping-Oriented Software Development Part I - Theoretical, Technical, and Organizational Aspects, **Institutsbericht Nr. 87.05, Institut für Informatik, Universität Zürich**.
- Pree, W. (1991): Object-Oriented Versus Conventional Construction of User Interface Prototyping Tools, **Dissertation, Johannes Kepler Universität Linz**.
- Rotz, B. v. (1987): Software Prototyping. **Diplomarbeit am Institut für Informatik** der ETH Zürich.
- Rumbaugh, J. (1991): **Object-Oriented Modeling and Design**, Prentice Hall, Englewood Cliffs. Sandberg, D.W. (1987): Smalltalk and Exploratory Programming, **ACM SIGPLAN Notices**, Vol. 22, No.10.
- Schmidt D., G. Pomberger, K. Bauknecht (1989): Prototyping-Oriented Software Development, Part II - The TOPOS Component Management System. **Institutsbericht Nr. 89.08, Institut für Informatik, Universität Zürich**.
- Servio (1991): **GemStone: Programming in OPAL**, Servio Corporation, Alameda CA.
- Shlaer, S., S. Mellor (1988): **Object-Oriented Systems Analysis**, Prentice Hall, Englewood Cliffs.
- Sommerville, I. (1992): **Software Engineering**, 4rd edition. Addison Wesley.
- Stachowiak, H. (1973): **Allgemeine Modelltheorie**. Springer Verlag, Wien.
- Standish, T.A., T. Taylor (1982): Initial thoughts on rapid prototyping techniques. **ACM Software Engineering Notes**, Vol. 7, No. 5, pp. 160-166.

- Swartout, W., R. Balzer (1982): On the inevitable intertwining of specification and implementation. **Communications of the ACM**, Vol. 25, No. 7, pp 438-440.
- Taenzer, D., M. Ganti, S. Podar (1989): Object-Oriented Reuse: The Yoyo Problem, **Journal of Object-Oriented Programming**, September/October, pp 30-45.
- Tichy, W. (1988): Tools for Software Configuration Management. In Winkler, J. (ed.) German Chapter of the ACM, Berichte 30, **Proceedings of the International Workshop on Software Version and Configuration Control**, Teubner, Stuttgart, pp 1-21.
- Tracz, W. (1987): Software reuse: Motivators and Inhibitors, **Proceedings of COMPCONS'87**, pp 358-363.
- Venken, R., M. Bruynooghe (1984): Prolog as a Language for Prototyping of Information Systems, in Budde, R. et al. (eds) **Approaches to Prototyping**, Springer Verlag, Berlin, pp 447-458.
- Wegner, P. (1987): Dimensions of Object-Based Language Design, **Proceedings of OOPSLA '87**, Orlando, Florida, special issue of ACM SIGPLAN Notices, Vol. 22, No. 12, pp 168-182.
- Weinand, A., E. Gamma, R. Marty (1989): Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. **Structured Programming**, Vol. 10, No 2.
- Winkler, J. (1988): **Proceedings of the International Workshop on Software Version and Configuration Control**, German Chapter of the ACM, Berichte 30, Teubner Stuttgart.
- Zehnder, C. A. (1987): **Informationssysteme und Datenbanken**, vdf Zürich.