Adaptierbare CASE-Werkzeuge in prozeßorientierten Software-Entwicklungsumgebungen

Vom Fachbereich Elektrotechnik und Informatik
der Universität-Gesamthochschule Siegen
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

von

Diplom-Ingenieur Marc Monecke

1. Gutachter: Prof. Dr. rer. nat. Udo Kelter
2. Gutachter: Prof. Dr. rer. nat. Volker Gruhn

Tag der mündlichen Prüfung: 7. Februar 2003

urn:nbn:de:hbz:467-303

Zusammenfassung

Die Entwicklung von Software-Systemen ist eine komplexe und anspruchsvolle Aufgabe, die bei großen Systemen meist auf mehrere Entwickler verteilt wird. *Software-Entwicklungsmethoden* strukturieren die Arbeit der Beteiligten: Sie legen Arbeitsschritte und Produkte fest, die in den einzelnen Schritten erzeugt oder verarbeitet werden. Produkte sind hier verschiedene Arten von *Software-Dokumenten*. Die Arbeit konzentriert sich auf die frühen Software-Entwicklungsphasen Analyse und Entwurf, in denen Anforderungen ermittelt und Modelle wichtiger Aspekte des Systems erstellt werden. In diesen Phasen kommen *Upper-CASE-Werkzeuge* zum Einsatz, mit denen die verschiedenen Arten von Software-Dokumenten bearbeitet, geprüft und in andere Formate transformiert werden können. Die Werkzeuge müssen also die im jeweiligen Dokument verwendeten Konzepte (wie Entitätstyp, Klasse, Vererbungsbeziehung) und die zugehörigen Notationen und Darstellungen unterstützen.

Die Anforderungen an CASE-Werkzeuge unterscheiden sich stark, da sie in unterschiedlichen Organisationen, von Personen mit unterschiedlichen Aufgaben und Zielen und zur Entwicklung unterschiedlicher Arten von Systemen eingesetzt werden – zum Teil sind die Anforderungen auch unbekannt, wenn in einem Projekt neue Wege beschritten werden.

Eine wichtige Anforderung an die Werkzeuge ist daher, daß sie an die jeweilige Einsatzsituation angepaßt werden können. Herkömmliche CASE-Werkzeuge erfüllen diese Anforderung nur unzureichend. *Meta-Umgebungen* erleichtern den Bau angepaßter Werkzeuge, allerdings konzentrieren sich die meisten bekannten Ansätze auf nur einen bestimmten Aspekt: Den Bau von CASE-Werkzeugen für einen gegebenen Dokumenttyp oder die Unterstützung des gegebenen Vorgehens- oder *Prozeßmodells*.

In der Arbeit wird ein Ansatz zum Bau von *prozeßorientierten Software-Entwicklungsumgebungen* entwickelt, mit dem CASE-Werkzeuge maßgeschneidert *und* mit dem Software-Entwicklungsprozeß integriert werden können. Die resultierenden Umgebungen nutzen ein *Object Management System* (OMS) als zentrales Repository.

Wichtige Merkmale sind die feingranulare Modellierung der Dokumente und eine Werkzeugarchitektur, mit der die Dienste des OMS zur Implementierung der Werkzeugfunktionen direkt ausgenutzt werden können. In der Arbeit wird ein Framework mit Werkzeugkomponenten entwickelt, das zum Bau mehrbenutzerfähiger und verteilt einsetzbarer Werkzeuge verwendet wird. Werkzeuge können einfach und schnell aus vorhandenen Komponenten zusammengesetzt und Parameter zur Steuerung dieser Komponenten festgelegt werden. Weiterhin können Komponenten erweitert und angepaßt werden, wodurch eine hohe Flexibilität bei der Adaptierung von Werkzeugen erreicht wird.

Zur Beschreibung des Prozeßmodells wird eine einfache Prozeßmodellierungssprache vorgeschlagen. Mit Framework-Komponenten kann eine Prozeßmaschine zur Steuerung und Überwachung des Prozesses gebaut und flexibel um prozeßspezifische Funktionen erweitert werden. Das Prozeßmodell dient hier als Spezifikation für die Prozeßmaschine. Die gebauten CASE-Werkzeuge werden mit dem Prozeß integriert, passen sich also an die aktuelle Prozeßsituation an und ermöglichen es dem Entwickler, den Prozeßfortschritt zu beeinflussen. Mit dem Bau von Planungs- und Managementwerkzeugen und ihrer Integration in die Umgebung wird die Flexibilität des Ansatzes geprüft.

Abstract

Software development is a complex and demanding task, frequently carried out by teams of multiple developers. *Software development methods* define the steps to be taken and the products to be maintained in order to attain a successful project outcome. Thus, methods outline the structure of a project. In the early phases, *upper-CASE tools* are used to edit, check, and transform various types of software documents. The requirements for these tools vary considerably, depending on the organization, user role, task, and kind of system to be developed. Consequently, tools have to be *adaptable* to the actual situation. Traditional CASE tools do not fulfill this demand sufficiently. *Meta environments* support tool developers in building customized tools. However, known systems are focused on only one aspect: The adaptation to different document types or the adaptation to different process models.

The thesis describes an approach for building *process-centered software development environments* (PSEEs). The resulting environments contain tools which are tailored to the given situation and are integrated with the software development process. An important component of these environments is an *Object Management System* (OMS). It serves as a central repository for both, software documents and information concerning the current process state. The OMS integrates the various tools with their different views on the common database and offers numerous services which are used for implementing tool functions – among them views, access controls, transactions, and distributed notifications.

Thereby, the costs for building meta environments and for building their instances, i.e. concrete environments, are reduced. The services mentioned can only be used if data models are *fine-grained* and if tools implement an *OMS-oriented* architecture. In the thesis, this architecture has been implemented in a framework. It is used to build distributed multi-user tools. The framework consists of components for building CASE tools and process tools. The configuration of tools depends on the data model of software documents and on additional information. These additional information are codified in *tool parameters* which are associated to type definitions in the data model. The resulting tool specification is called *tool schema*. Besides this *black-box* reuse, tool components can be extended with tool-specific functionality. This *white-box* reuse requires a more extensive knowledge and offers more flexible means for tool adaptation.

For the description of process models we define a simple process modeling language which is used for demonstration purposes only. Based on the framework, a process engine can be built. It controls process performance and can be flexibly extended with process-specific functionality. CASE tools adapt themselves to the current process situation and allow their users to influence process performance. Additionally, tools for planning and managing software processes have been built and have been integrated into the environment to underline the flexibility of the approach.

Vorwort

Die vorliegende Dissertation habe ich als wissenschaftlicher Mitarbeiter der Fachgruppe Praktische Informatik an der Universität Siegen verfaßt.

Ich möchte an dieser Stelle meinem Chef und Erstgutachter Prof. Dr. Udo Kelter danken für seine Betreuung und für die Schaffung der angenehmen Rahmenbedingungen. Auch gilt mein Dank Prof. Dr. Volker Gruhn für die Übernahme des Zweitgutachtens.

Ich danke allen jetzigen und ehemaligen Kolleginnen und Kollegen des Fachbereichs Informatik, die mir stets mit Rat und Tat zur Seite standen, sowie den engagierten Studenten, die einen Beitrag zur Implementierung und Validierung des beschriebenen Ansatzes geleistet haben.

Schließlich gilt mein herzlicher Dank meinen Eltern für ihre vielfältige Unterstützung und Hilfe.

Nauroth, im Februar 2003

Marc Monecke

Besides, the determined Real Programmer can write
Fortran programs in any language.

(Ed Post, Datamation, 1983)

Inhaltsverzeichnis

Kapitel 1

Einleitung, Motivation und Übersicht

Die Entwicklung komplexer Software-Systeme ist heute ohne Werkzeugunterstützung nicht mehr denkbar [156]. Der Werkzeugkasten des Software-Entwicklers enthält Werkzeuge für die Analyse, den Entwurf, die Implementierung und die Qualitätssicherung von einzelnen Software-Komponenten und ganzen Software-Systemen. Neben diesen Produktionswerkzeugen tragen Werkzeuge zur Planung, Steuerung und Kontrolle der Entwicklungstätigkeiten zur erfolgreichen Durchführung eines Projekts bei: Sie helfen, Aufwände, Kosten und Termine zu überwachen und die Qualität des Entwicklungsprozesses zu sichern.

In den frühen Software-Entwicklungsphasen kommen *Upper-CASE-Werkzeuge* [253] zum Einsatz, im folgenden kurz als CASE-Werkzeuge bezeichnet. Sie unterstützen die Entwickler beim Erzeugen, Bearbeiten, Prüfen, Analysieren und Transformieren von Software-Spezifikationen im Rahmen der Analyse und des Entwurfs von Softwaresystemen.

1.1 Methoden und Werkzeuge

Bei der Entwicklung eines Software-Systems nutzen die Entwickler CASE-Werkzeuge, um Dokumente zu erzeugen und zu bearbeiten. Welche Arten von Dokumenten, also welche *Dokumenttypen* dies sind, und in welchen Schritten bei der Erstellung und Bearbeitung vorgegangen wird, hängt von der *Software-Entwicklungsmethode* ab, die die Entwickler verwenden. In vielen Fällen werden Entscheidungen bei der Entwicklung ad hoc getroffen: Welche Aspekte des zu entwickelnden Systems modelliert werden, in welchen Schritten vorgegangen werden soll, und wer welche Aufgaben übernimmt, hängt ab von den gesammelten Erfahrungen der Beteiligten und deren Kenntnissen; davon, welche Entwickler oder Werkzeuge verfügbar sind und wie beide sich in vergangenen Projekten bewährt haben. Oder auch davon, welcher Zeit- und Kostenrahmen der Entwicklung gesetzt ist. Die Fragmente der verwendeten Methode sind also nur implizit in den Köpfen der Beteiligten definiert – und Konsistenzprobleme zwischen den verschiedenen Ausprägungen wahrscheinlich.

In einem *Methodenhandbuch* wird die Methode explizit beschrieben. Ausgangspunkt kann eine Beschreibung der Methode aus einem Lehrbuch sein, die allerdings meist stark erweitert und verfeinert werden muß, bevor sie als Anleitung für die Entwickler dienen kann [296]. Somit werden im Methodenhandbuch wichtige Entscheidungen festgehalten und sind für alle

Beteiligten zugänglich. Durch die Anpassung und Erweiterung der Methodendefinition kann auch der Lernprozeß innerhalb einer Organisation dokumentiert werden. Die Methodendefinition ist also nicht statisch, sondern muß an konkrete Anforderungen und Randbedingungen adaptiert werden können [180]. Wichtig ist daher auch, daß die Entwickler einfach auf die Methodendefinition zugreifen können. Sowohl die Anpassung als auch die Bereitstellung der Methodendefinition am Arbeitsplatz des Entwicklers ist bei Methodenhandbüchern jedoch schwierig [170]. Gerade die wachsende Zahl neuer Anwendungsgebiete wie *E-Commerce* und *Pervasive Computing* wird in Zukunft den Bedarf an neuen und angepaßten Methoden weiter erhöhen [70]. Der Mobilfunk-Hersteller Nokia setzt nach eigenen Angaben (zitiert in [187]) mehr als 150 verschiedene maßgeschneiderte Methoden ein, angepaßt an die jeweilige Projektsituation.

Durch die Verbreitung der *Unified Modelling Language* (*UML*) [265] wird die Situation zum Teil entschärft, da nun eine standardisierte Sprache zur Verfügung steht, die von zahlreichen Werkzeugherstellern unterstützt wird. Allerdings ist die UML so umfangreich, daß einerseits viele Werkzeuge nur einen Teil der Sprache unterstützen; andererseits in einem konkreten Projekt meist nur eine Teilmenge der UML sinnvoll anwendbar ist. Eine solche Einschränkung wird aber von den Werkzeugen nicht unterstützt; ebensowenig ein definiertes Vorgehen bei der Anwendung der UML. Oft wird die UML als Basis für eine spezielle Sprache genutzt (etwa zur Beschreibung von Produktfamilien [57], Frameworks [118] oder von Software-Prozessen [178]). Ausgenutzt wird hierbei der hohe Bekanntheitsgrad von UML und die Tatsache, daß UML-Werkzeuge verfügbar sind. Die spezielle Semantik der Sprache muß jedoch zusätzlich definiert werden und ist Benutzern wie Werkzeugen zunächst unbekannt. Somit können die Werkzeuge keinerlei Unterstützung anbieten; das Erweitern der Werkzeuge um spezielle Darstellungen, Prüf- und Transformationskommandos ist meist nicht oder nur sehr aufwendig möglich. Die Anforderungen an die Werkzeuge werden durch die Methodendefinition vorgegeben.

Eine *Methodendefinition* besteht aus drei Teilen:

1. Einer Menge von *Konzepten*, die das Wissen repräsentieren, das in der Methode erfaßt, manipuliert und gespeichert wird. Beispiele sind Klassen, Attribute, Operationen, Assoziationen.
2. Einer Menge von *Notationen*, mit denen das Wissen für die verschiedenen Teilnehmer im Entwicklungsprozeß aufbereitet wird, etwa Klassendiagramme oder die baumartige Darstellung einer Paketstruktur.
3. Einer Menge von *Vorgehensweisen* oder Prozessen, die festlegen, in welchen Schritten bei der Entwicklung vorgegangen wird, welche Arten von Dokumenten als Ein- und Ausgaben für die einzelnen Schritte dienen, und welche Anforderungen an die ausführenden Personen gestellt werden.

Im *Diamond Model* [180] in Abbildung 1.1 sind die Teile der Methodendefinition und damit die verschiedenen Aspekte einer Methode dargestellt: Die Auswahl von Konzepten, Notationen und Prozessen wird beeinflußt durch die Ziele oder Randbedingungen, die im Projekt berücksichtigt werden müssen. Die Linien zwischen den Knoten verdeutlichen deren Abhängigkeiten. Jeder Aspekt kann nur zum Teil unabhängig betrachtet werden und muß schließlich mit den übrigen Aspekten integriert werden.

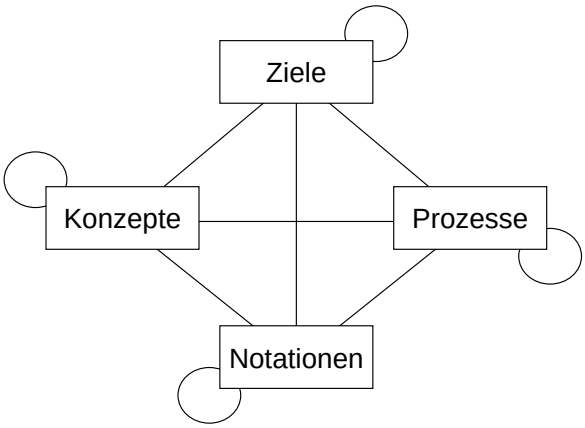


Abbildung 1.1: *Diamond Model*

1.1.1 Meta-Modell

Die Eigenschaften einer Methode können formal in einem *Meta-Modell* [187] definiert werden: Während die Anwendung einer Methode ein *Modell* für das zu entwickelnde System hervorbringt (etwa in Form von Daten- und Funktionsmodellen), beschreibt das *Meta*-Modell die Eigenschaften der Methode. Ein *Dokumenttyp* beschreibt die Eigenschaften von Dokumenten. Er faßt eine Teilmenge der Konzepte und Notationen zusammen, die in der Methode verwendet werden. Beispiele für Dokumenttypen sind ER-Diagramm, Datenlexikon, Datenflußdiagramm oder Zustandsdiagramm. Mit *Konsistenzbedingungen* wird die Korrektheit einzelner Dokumente oder mehrerer zusammengehörender Dokumente definiert. Das abstrakte Meta-Modell aus Abbildung 1.2 faßt die verschiedenen Aspekte zusammen (vgl. [256]).

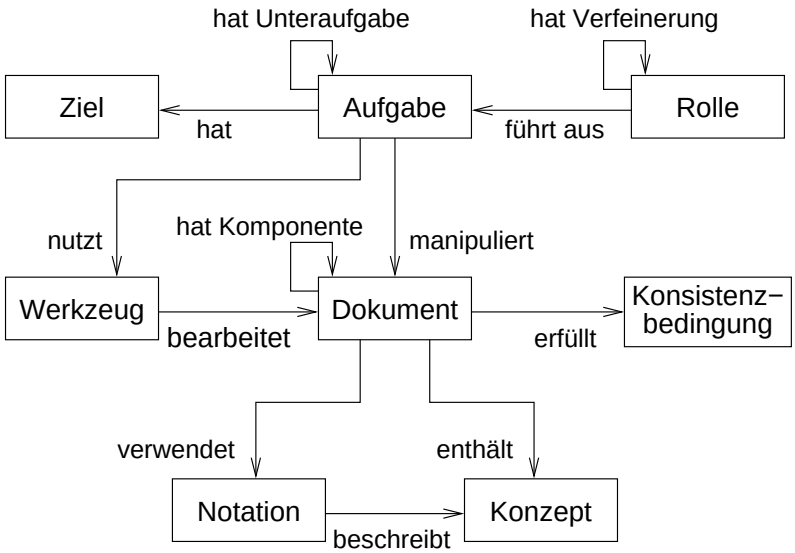


Abbildung 1.2: Meta-Modell einer Software-Entwicklungsmethode

Das Forschungsgebiet des *Method Engineering* [39] beschäftigt sich mit der Frage, wie mit Hilfe von Meta-Modellen konkrete Methoden spezifiziert und eine Werkzeugunterstützung

für diese Methoden gebaut werden kann [309]. Oft muß dabei die konkrete Projektsituation berücksichtigt werden: *Context-Specific Method Engineering* [284]. Werkzeuge für den Methodenentwurf werden auch als *Computer Aided Methodology Engineering*-Werkzeuge (CAME-Werkzeuge) [238] bezeichnet.

1.1.2 CASE-Werkzeuge

CASE-Werkzeuge sollen die Entwickler bei ihren Aufgaben unterstützen und zum Teil auch Arbeitsschritte automatisieren, etwa bei der Versionierung von Dokumenten [121]. Die Funktionen von CASE-Werkzeugen lassen sich unterteilen in Funktionen zur Darstellung, Manipulation, Analyse und Transformation von Dokumenten – zusammengefaßt als *Production Technology*. Die *Coordination Technology* umfaßt Funktionen, mit denen Aufgaben definiert, den Entwicklern zugewiesen und die Zusammenarbeit der Entwickler unterstützt wird [163].

Die Werkzeuge, die in einem Projekt eingesetzt werden, sollten miteinander *integriert* sein. Eine Sammlung integrierter Werkzeuge, die eine bestimmte Phase im Software-Lebenszyklus unterstützen, wird als *CASE-Werkbank* bezeichnet; die Unterstützung für den gesamten Software-Lebenszyklus als *CASE-Umgebung*. Die Integration von Werkzeugen hat verschiedene Ziele, darunter die einfachere Bedienung durch eine gleichartige Benutzungsschnittstelle, die Kontrolle von Diensten verschiedener Werkzeuge durch die Umgebung und den Austausch oder die gemeinsame Nutzung von Daten.

Zwischen einer Methode und den CASE-Werkzeugen bestehen zwei Arten von Beziehungen: Zum einen definiert die Methode Anforderungen an die Werkzeuge. Die Werkzeuge müssen die Entwickler beim Ausführen der verschiedenen Arbeitsschritte oder *Aufgaben* unterstützen, also die Bearbeitung der Dokumente verschiedenen Typs ermöglichen und Funktionen zur Gewährleistung der Konsistenz anbieten. Außerdem müssen es die Werkzeuge ermöglichen, Ergebnisse einer Aufgabe in andere Aufgaben zu übernehmen und dort weiterzubearbeiten, somit die einzelnen Aufgaben zum Prozeß zu verbinden.

Zum anderen legt die Methode fest, zu welchem Zeitpunkt ein Werkzeug eingesetzt wird, auf welchen Dokumenten es arbeitet, und welche Voraussetzungen die Benutzer erfüllen müssen. Diese Voraussetzungen, Fähigkeiten, Kenntnisse und Verantwortlichkeiten werden in *Rollen* zusammengefaßt und den Benutzern zugeordnet. Rollen können auch hierarchisch sein, etwa die Rolle des Programmierers mit ihren Verfeinerungen C-Programmierer und Java-Programmierer. Es ist daher sinnvoll, Werkzeuge in einer Umgebung zu verwalten, die auch die Zuordnung von Benutzern, Aufgaben, Dokumenten und Werkzeugen übernimmt.

Nutzung der Werkzeuge

CASE-Werkzeuge haben positive Auswirkungen auf die Produktivität und Qualität der Software-Entwicklung [173] und fördern ein formales Vorgehen [222]. Voraussetzung dafür ist natürlich, daß die Werkzeuge tatsächlich genutzt werden. Lending und Chervany haben die Nutzung von CASE-Werkzeugen im Jahre 1998 untersucht und in [222] auch ältere Forschungsergebnisse zitiert. Sie kommen zu folgendem Ergebnis:

– Ein nicht zu vernachlässigender Anteil der Firmen und Organisationen, die Software entwickeln, nutzen keine CASE-Werkzeuge.

– Ein nicht zu vernachlässigender Anteil der Firmen und Organisationen, die CASE-Werkzeuge angeschafft haben, brechen die Nutzung nach kurzer Zeit ab.
– Innerhalb der Firmen oder Organisationen, die CASE-Werkzeuge einsetzen, gibt es einen nicht zu vernachlässigenden Anteil von Entwicklern, die die Werkzeuge nicht nutzen.
– Ein nicht zu vernachlässigender Anteil der Funktionen des Werkzeugs wird nicht genutzt. Hierzu zählen oft Funktionen zur Analyse und Transformation von Dokumenten.

Zu den Gründen für diese Situation gehören sicher die folgenden:

– CASE-Werkzeuge und CASE-Umgebungen sind sehr komplexe Systeme. Die hohe Komplexität verstellt den Blick auf den tatsächlichen Nutzen der Werkzeuge [173], schreckt die Benutzer ab und macht die Werkzeugeinführung aufwendig und teuer.
– Die Einführung von CASE-Werkzeugen hat nicht unmittelbar die gewünschte und erwartete Produktivitätssteigerung zur Folge. Zum Teil sinkt die Produktivität zunächst ab, wodurch die Erwartungen der Nutzer enttäuscht werden [210].
– Entwickler sind nicht sehr stark motiviert, verfügbare Werkzeuge auch tatsächlich zu verwenden, da sie keinen unmittelbaren Nutzen erkennen [222]. Dem kann durch Vorschriften des Managements entgegengewirkt werden.

1.1.3 Fazit

Software-Entwicklungsmethoden definieren einen Rahmen, in dem Projekte ausgeführt werden: Sie schreiben das Vorgehen und die zu produzierenden Dokumente mit ihren Eigenschaften vor und tragen so zur Qualitätssicherung bei. Modelle und Standards zur Erhöhung der Qualität von Software-Produkten und Software-Prozessen wie das *Capability Maturity Model* [168] oder der ISO-Standard 9001 [181] fordern daher ein definiertes und wiederholbares Vorgehen bei der Software-Entwicklung.

Durch die Definition von Aufgaben und ihren Ergebnissen wird die Transparenz des Entwicklungsprozesses erhöht; durch die Berücksichtigung von Erfahrungen aus vergangenen Projekten wird der Reifungsprozeß von Methode und Organisation unterstützt. Entscheidend ist, daß die Methode an die tatsächlichen Anforderungen angepaßt ist, also die Art des Projektes sowie organisatorische und technische Randbedingungen berücksichtigt, und den Entwicklern bei ihrer Arbeit direkt zugänglich ist.

Die CASE-Werkzeuge, die im Projekt zum Einsatz kommen, müssen wiederum an die Methode angepaßt sein, also die verwendeten Dokumenttypen mit ihren Konzepten, Notationen, Darstellungen, Konsistenzbedingungen und Transformationsbeziehungen unterstützen. Und Sie müssen untereinander integriert sein. Erfüllt ein Werkzeug nicht die Anforderungen, kann es nicht effizient genutzt werden, weil Dokumente manuell nachbearbeitet werden müssen oder wichtige Eigenschaften des zu entwickelnden Systems nicht passend modelliert werden können. Die Entwickler müssen also entweder ihre Arbeitsweise an das Werkzeug anpassen – oder auf eine Werkzeugunterstützung ganz verzichten [232, 303]. Dahanayake schreibt dazu im Jahr 2001: *"Lack of flexibility has been a great drawback. The idea of providing a tailorable, configurable environment, which is customized as necessary for different organizations, projects, and individuals, has not been achieved."* [70]

Die Anpassung der Werkzeuge an die gegebenen Einsatzbedingungen [312] kann also stark dazu beitragen, daß die Werkzeuge auch tatsächlich genutzt werden. Weiterhin ist es sinnvoll, die Komplexität der Werkzeuge möglichst gering zu halten. Dies erleichtert die Einarbeitung und erhöht die Bereitschaft zur Nutzung.

Schließlich muß die Möglichkeit bestehen, die Ausführung von Funktionen, die für den weiteren Fortschritt im Entwicklungsprozeß relevant sind, überwachen zu können. So kann sichergestellt werden, daß die Konsistenz eines Dokuments geprüft und das Dokument fehlerfrei ist, bevor es an eine Folgeaufgabe weitergeleitet wird, oder daß abgeleitete Dokumente erzeugt wurden, die eine Folgeaufgabe benötigt.

Zu klären bleibt die Frage, wie solche Werkzeuge effizient gebaut und in einer CASE-Umgebung integriert werden können.

1.2 Meta-Umgebungen

Meta-Umgebungen reduzieren den Aufwand und damit die Kosten für den Bau von CASE-Werkzeugen: Sie bieten mächtigere Konstrukte als Programmiersprachen an und erlauben den Bau von CASE-Werkzeugen auf einer höheren Abstraktionsebene. Zu den Ansätzen für Meta-Umgebungen zählen [195]:

1. *Adaptierbare Umgebungen* (*customizable environments*) [238], die aus einem Kern mit generischen Funktionen bestehen. Diese Funktionen sind hinreichend allgemein, um in vielen Werkzeugen im betrachteten Anwendungsbereich genutzt werden zu können. Umgeben wird dieser generische Kern von einer adaptierbaren Schale, die die Besonderheiten der verschiedenen Werkzeuge berücksichtigt, also das konkrete Datenmodell der Dokumente und die gewünschten Darstellungen.
2. Ansätze zur *Prozeßmodellierung* [55, 66, 79], mit denen Software-Entwicklungsprozesse beschrieben und in *prozeßorientierten Umgebungen* (PSEU) [54, 122] ausgeführt werden können. Die PSEU schafft die Verbindung zwischen CASE-Werkzeugen und dem Software-Entwicklungsprozeß.
3. Ansätze zur *Werkzeugintegration* [305], die vorhandene Werkzeuge zu einer Umgebung kombinieren. Eine erfolgreiche Integration setzt allerdings voraus, daß diese bereits bei der Entwicklung der Werkzeuge eingeplant war.

Die unterschiedlichen Ansätze verfolgen unterschiedliche Ziele und haben unterschiedliche Vor- und Nachteile: Bei adaptierbaren Umgebungen steht die Anpassung an Konzepte und Notationen im Vordergrund; Vorgehensweisen oder Prozesse werden hier nur selten unterstützt. Ansätze zur Prozeßmodellierung konzentrieren sich hingegen auf die Beschreibung und Ausführung der Prozesse, lassen aber Konzepte und Notationen außer acht [232]. Die Werkzeugintegration zielt eher auf die Infrastruktur, die zum gemeinsamen Betrieb mehrerer Werkzeuge gebraucht wird.

Jeder Ansatz konzentriert sich also auf ein bestimmtes Teilproblem und erzeugt eine Insellösung; Brücken zwischen den verschiedenen Lösungen existieren aber meist nicht [180]. Gemein ist den verschiedenen Ansätzen, daß sie ein *Konstruktionsmodell* [195] definieren. Die Instanzen des Konstruktionsmodells sind *Umgebungsspezifikationen*, die wiederum in

konkrete Umgebungen transformiert werden können. Ein *Konstruktionsprozeß* definiert die nötigen Schritte zum Bau einer Umgebung (vgl. Abbildung 1.3).

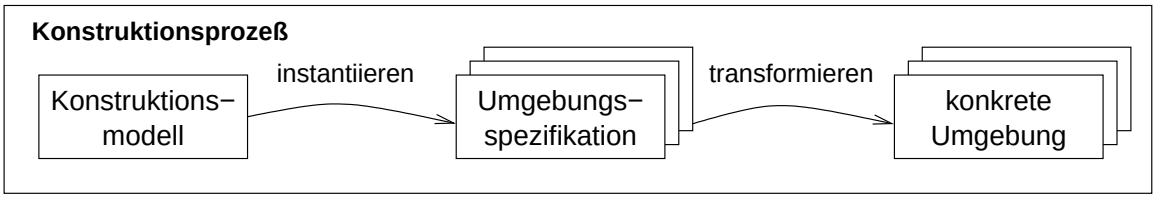


Abbildung 1.3: Konstruktionsprozeß in Meta-Umgebungen

Anforderungen an eine Meta-Umgebung

Meta-Umgebungen sind natürlich auch selbst komplexe Software-Systeme, die zahlreiche Anforderungen erfüllen müssen [195]. Die erste Anforderung ergibt sich aus dem vorherigen Abschnitt:

1. Die Meta-Umgebung muß eine Anpassung der Werkzeuge an Konzepte, Notationen und Vorgehen der gewählten Methode ermöglichen und die Integration der erzeugten Werkzeuge unterstützen.
2. Die produzierten Werkzeuge müssen Qualitätsanforderungen an Geschwindigkeit, Benutzungsschnittstelle und Funktionsumfang erfüllen. Der benötigte Funktionsumfang hängt vom Einsatz der Werkzeuge ab und kann Funktionen zur Konsistenzsicherung, Mehrbenutzerfähigkeit, Verteilung und Zugriffsschutz umfassen.
3. Die Meta-Umgebung muß einerseits einfach verwendbar sein, so daß CASE-Werkzeuge einfach und schnell gebaut werden können. Dazu müssen auch Mechanismen vorhanden sein, die die Wiederverwendung von Werkzeugteilen ermöglichen sowie Fehler und Konsistenzverletzungen finden. Andererseits muß die Meta-Umgebung hinreichend flexibel sein, so daß bekannte wie zukünftige Anforderungen auch berücksichtigt werden können.

1.3 Ziele der Arbeit

Der erwartete Nutzen einer Meta-Umgebung besteht also darin, zur Steigerung der Produktivität und Qualität in der Software-Entwicklung beizutragen: Sie ermöglicht es, mit geringem Aufwand Werkzeuge herzustellen, die die verwendete Methode unterstützen und somit effizient einsetzbar sind. Auch sollten die Entwickler stärker motiviert sein, maßgeschneiderte Werkzeuge einzusetzen, so daß die positiven Auswirkungen von CASE-Werkzeugen durch die breitere Nutzung weiter verstärkt werden.

Natürlich stehen dem erwarteten Nutzen von Meta-Umgebungen recht hohe Kosten gegenüber. Sie entstehen bei der Entwicklung der Meta-Umgebung, der Einarbeitung der Werkzeugentwickler und dem anschließenden Bau der eigentlichen Werkzeuge. Das Verhältnis zwischen Kosten und Nutzen kann zugunsten der Meta-Umgebungen verschoben werden, wenn

- die Kosten für den Bau der Meta-Umgebung wie für den Bau der Werkzeuge reduziert werden können.
- die Meta-Umgebung und die gebauten Werkzeuge möglichst häufig eingesetzt werden, so daß sich der initiale Aufwand lohnt.
- die Anforderungen an die benötigten CASE-Werkzeuge sehr stark von den Leistungen am Markt verfügbarer Werkzeuge abweichen, somit der Einsatz letzterer eine starke Einschränkung und Behinderung der Entwickler zur Folge hätte.
- berücksichtigt wird, daß Auswahl, Evaluierung, Konfiguration und Einsatz kommerzieller Werkzeuge ebenfalls aufwendig und teuer sind [43, 277].

Bei Meta-Umgebungen muß ein Kompromiß gefunden werden zwischen Spezialisierung und Allgemeingültigkeit [195]: Spezialisiert sich die Umgebung auf einen bestimmten Anwendungsbereich, kann sie für diesen eine besonders effiziente Unterstützung anbieten. Allerdings ist ihre Einsetzbarkeit beschränkt. Umgekehrt garantiert eine große Allgemeingültigkeit den vielfältigen Einsatz, allerdings ist die Unterstützung beim Werkzeugbau geringer und damit sind die Kosten höher.

Genauso versuchen kommerzielle CASE-Produkte, eine möglichst große Schnittmenge der Anforderungen unterschiedlicher Nutzer zu erfüllen. Weichen die Anforderungen einer konkreten Organisation oder eines bestimmten Projekts stark von diesen ab, so handelt es sich meist um einen eher seltenen Sonderfall. Hier lohnt sich der Bau maßgeschneiderter Werkzeuge aus Sicht der Anforderungen, oft jedoch nicht in Anbetracht der geringen Einsatzmöglichkeiten.

Die Arbeit konzentriert sich auf die technischen Aspekte von Meta-Umgebungen und schlägt eine Architektur zum Bau von Meta-Umgebungen vor. Auf Basis dieser Architektur wird eine Meta-Umgebung konstruiert, die versucht, die verschiedenen Ansätze zu kombinieren, also sowohl eine Anpassung an Konzepte und Notationen zu ermöglichen (wie adaptierbare Umgebungen), als auch das individuelle Vorgehensmodell zu unterstützen (wie Prozeßmodellierungsansätze) und integrierte Werkzeuge zu produzieren (wie die Integrationsansätze).

Sowohl beim Bau der Meta-Umgebung als auch beim Bau der CASE-Werkzeuge mit dieser Meta-Umgebung sollen die folgenden softwaretechnischen Prinzipien berücksichtigt werden:

1. *Kosten*: Kosten und Aufwand für den Bau von Meta-Umgebung und Werkzeugen sollen möglichst gering sein.
2. *Wiederverwendung*: Beim Bau der Meta-Umgebung wie beim Werkzeugbau sollen viele Komponenten wiederverwendet werden. Komponenten sind hier sowohl Bausteine, die als Teile von Meta-Umgebung oder Werkzeugen entwickelt wurden, als auch genutzte Basissysteme.
3. *Wartbarkeit*: Die häufige Verwendung einzelner Komponenten erhöht die Wartbarkeit. Sie wird auch durch eine einfache Architektur gefördert.
4. *Flexibilität und Erweiterbarkeit*: Meta-Umgebung und Werkzeuge sollen flexibel erweiterbar sein, so daß neue Anforderungen an die angebotenen Darstellungen und Funktionen auch nachträglich berücksichtigt werden können.
5. *Konsistenz*: Die Meta-Umgebung soll den Bau von fehlerfreien Werkzeugen unterstützen und dazu Mechanismen zum Feststellen von Fehlern und Inkonsistenzen in der Werkzeugspezifikation anbieten.

1.4 Bekannte Ansätze

Es existieren bereits zahlreiche Lösungsvorschläge für die beschriebenen Teilprobleme. In der Arbeit werden die verschiedenen Ansätze ausgewertet, geeignete angepaßt und zu einer Lösung des Gesamtproblems kombiniert. Die folgenden Abschnitte gehen kurz auf einige bekannte Ansätze ein.

1.4.1 Werkzeugbau

Ansätze zum Bau von CASE-Werkzeugen werden unter dem Begriff *Meta-CASE* [4] zusammengefaßt. Meta-CASE-Systeme werden auch als *CASE shells* bezeichnet. Sie enthalten Mechanismen zur Konstruktion von Werkzeugen für beliebige Methoden [44], beschränken sich aber meist auf deren Konzepte und Notationen.

Frameworks

Objektorientierte Frameworks enthalten Werkzeugkomponenten und definieren Beziehungen zwischen diesen Komponenten. Beim Bau eines konkreten Werkzeugs werden Framework-Komponenten so erweitert, daß sie die werkzeugspezifischen Eigenschaften realisieren. *Anwendungsframeworks* [327] sind *generische Anwendungen* [125]: Sie enthalten eine softwaretechnische Architektur und die für den Anwendungsbereich relevanten fachlichen Abstraktionen. Ein Anwendungsframework für graphische Editoren ist *ET++* [127]; die Weiterentwicklung *MET++* [316] zielt auf Multimedia-Anwendungen. Mit *MetaMOOSE* [110] werden CASE-Werkzeuge gebaut. Mit *MViews* [147] können Werkzeuge mit verschiedenen Sichten auf die Software-Dokumente realisiert und auch CSCW-Funktionen integriert werden [149]. Frameworks sind meist sehr flexibel in der Anwendung, setzen aber einen großen Einarbeitungsaufwand und Programmierkenntnisse voraus.

Anwendungsgeneratoren

Ein *Anwendungsgenerator* erzeugt aus einer Werkzeugspezifikation ein lauffähiges Werkzeug. Die Implementierung des Werkzeugs basiert dann meist auch auf einem Framework, auf das der Werkzeugentwickler aber nicht direkt zugreift. Beispiele für Anwendungsgeneratoren sind *JANUS* [18] für textuelle Werkzeuge und der von Gille [129] beschriebene Generator für graphische Editoren. Auch kommerzielle Meta-CASE-Systeme wie *TBK/Toolbuilder* [5] und *MetaEdit+* [197] arbeiten mit Werkzeugspezifikationen. Anwendungsgeneratoren sind sehr komplexe Systeme, die die Spezifikation von Werkzeugen mit mächtigen Konstrukten und auf hoher Abstraktionsebene erlauben – vorausgesetzt, der Werkzeugentwickler beherrscht die proprietäre Spezifikationssprache. Problematisch ist ihre Inflexibilität, da Erweiterungen eine Änderung des Frameworks, der Spezifikationssprache und des Generators erfordern können.

Interpreter-Ansätze

Bei *Interpreter-Ansätzen* wird nicht der Anwendungsquelltext aus einer Spezifikation erzeugt, sondern die Spezifikation zur Laufzeit von einer generischen Anwendung interpretiert [91]. Viele 4GL-Systeme [76] fallen in diese Kategorie. Die Interpreter ähneln also in ih-

ren Vor- und Nachteilen stark den Generatoren; lediglich die Entwicklungszyklen sind hier kürzer, dafür ist die Geschwindigkeit der resultierenden Werkzeuge meist geringer.

1.4.2 Prozeßorientierte Software-Entwicklungsumgebungen

Prozeßorientierte Software-Entwicklungsumgebungen (PSEU) sind sowohl Meta-Umgebungen als auch konkrete Umgebungen: Als Meta-Umgebung erlauben sie die Beschreibung und Analyse des Prozesses. Die Prozeßbeschreibung, das *Prozeßmodell*, dient dann in der konkreten Umgebung zur Führung der Entwickler durch den Prozeß. Aktive Komponente einer PSEU ist die *Prozeßmaschine*, die auf Basis des Prozeßmodells den Fortschritt im Prozeß steuert und überwacht.

Die Unterschiede zwischen verschiedenen PSEU liegen in der verwendeten Prozeß-Modellierungssprache, den verfügbaren Werkzeugen zur Interaktion mit dem Prozeß und in der Art, wie CASE-Werkzeuge in die PSEU integriert werden können [8, 262].

In dieser Arbeit ist besonders der letzte Punkt von Interesse: Erst durch eine gute Integration von CASE-Werkzeugen und Prozeß wird das in der Methode definierte Vorgehensmodell für die Entwickler direkt zugänglich und es kann sich unmittelbar auf ihre Tätigkeit auswirken. CASE-Werkzeuge und Prozeß interagieren in zwei Richtungen:

Die aktuelle Prozeßsituation beeinflußt die Werkzeuge – jedem Prozeßteilnehmer müssen in einer gegebenen Prozeßsituation die passenden Werkzeuge zur Verfügung stehen und er muß auf die Dokumente zugreifen können, die er zur Erledigung seiner Aufgabe braucht. Bei einer weitergehenden Integration wirkt sich die Prozeßsituation auch auf die Eigenschaften der Werkzeuge aus und beeinflußt die verfügbaren Kommandos und Darstellungen. Umgekehrt soll der Entwickler über sein CASE-Werkzeug den weiteren Prozeßfortschritt beeinflussen können. Änderungskommandos des Werkzeugs müssen sich also auf den Prozeßzustand auswirken können.

Die Prozeßintegration der Werkzeuge erleichtert den Entwicklern die Interaktion mit dem Prozeß, weil keine zusätzlichen Werkzeuge benutzt und keine zusätzlichen Arbeitsschritte ausgeführt werden müssen, um die Prozeßmaschine über durchgeführte Änderungen zu informieren. Außerdem wird die Konsistenz zwischen den Tätigkeiten der Entwickler und dem Prozeßmodell erhöht, da mit den Werkzeugen nur erlaubte Aktionen durchgeführt werden können.

Zwei Ansätze zur Integration von Werkzeugen existieren:

1. Ein werkzeugspezifischer *Adapter* vermittelt zwischen PSEU und Werkzeug [88]: Um ein Werkzeug zu starten, ruft die PSEU Operationen des Adapters auf, der Adapter startet das passende Werkzeug mit den nötigen Parametern. Ist die Sitzung beendet, liefert der Adapter das Ergebnis der Werkzeugausführung an die PSEU zurück. PSEU und Werkzeug können also nicht während der Werkzeugausführung miteinander kommunizieren.
2. Bei einer *nachrichtenbasierten Integration* kommunizieren PSEU und Werkzeug über eine Kommunikationsinfrastruktur [280]. Somit können einzelne Werkzeugdienste von der PSEU aufgerufen werden und umgekehrt kann das Werkzeug Rückmeldungen über die Ausführung einzelner Kommandos liefern. Die Implementierung dieser Kommunikation ist aber sowohl im Werkzeug als auch in der PSEU aufwendig und fehleranfällig und wird

von kommerziellen Werkzeugen selten unterstützt. Ein funktionierendes Beispiel beschreiben Garg et al. in [128].

Alternativ zur nachträglichen Integration vorhandener Werkzeuge können auch Werkzeuge gebaut werden, die auf den Prozeß zugeschnitten sind. Ein Beispiel hierzu wurde im *Goodstep*-Projekt [100] erarbeitet und im kommerziellen Umfeld validiert [99]. Andere Ansätze sind von Nuseibeh [258] und Pohl et al. [275] beschrieben worden. Die bessere Prozeßintegration wird hier mit einem hohen Aufwand für die Werkzeugentwicklung, eingeschränkten Werkzeugfunktionen und geringer Flexibilität bei der Anpassung und Erweiterung von Werkzeugen erkauft.

1.4.3 Vollständige Methodenunterstützung

Eine vollständige Methodenunterstützung kann nur erreicht werden, wenn Konzepte, Notationen *und* Vorgehen der Methode in den Werkzeugen berücksichtigt werden. Mit den bisher beschriebenen Ansätzen ist das nicht möglich.

ISTAR [88] ist ein frühes Beispiel für Meta-Umgebungen, mit denen eine vollständige Methodenunterstützung möglich war. ISTAR konzentrierte sich allerdings eher auf die späten Software-Entwicklungsphasen und war in Varianten für die Entwicklung in C, Pascal und Ada verfügbar.

Marttiin beschreibt in [236], wie mit *MetaEdit+* ein Editor für Prozeßmodelle gebaut wurde: Das Meta-Prozeßmodell wird mit den Möglichkeiten des Meta-CASE-Systems beschrieben und den Aufgabentypen werden Werkzeuge zugeordnet. Mit dem so entstandenen Editor können dann konkrete Prozesse spezifiziert und die Werkzeuge gestartet werden. Allerdings gibt es keine Prozeßmaschine, und die Werkzeuge selbst sind nicht mit dem Prozeß integriert.

Lyytinen et al. [232] erweitern *MetaEdit+* um eine Prozeßunterstützung: Die vorher getrennten Meta-Modelle zur Beschreibung von Dokumenten aus *MetaEdit+* und zur Beschreibung von Prozessen aus *PRIME* [273] werden über Beziehungen verknüpft und somit Konzepte und Notationen den Aufgaben im Prozeßmodell zugeordnet. Die Werkzeuge werden an die Anforderungen der Aufgaben angepaßt, reagieren allerdings selbst nicht auf die aktuelle Prozeßsituation: Eine solche Anpassung ist in den *MetaEdit+*-Werkzeugen nicht vorgesehen; auch werden Prozeßzustand und Dokumente separat verwaltet. Das resultierende Meta-Modell ist sehr komplex und die Werkzeugspezifikation mithin sehr kompliziert, da Werkzeugeigenschaften nicht zu sinnvollen 'Konfigurationen' zusammengefaßt werden können.

In einem Prototypen namens *Viewer* [258] werden die Sichtweisen der verschiedenen Prozeßteilnehmer (*view points*) [260] explizit modelliert. Jeder *view point* enthält Dokumente, Werkzeuge und den relevanten Ausschnitt aus dem Prozeßmodell. Allerdings handelt es sich um sehr einfache Werkzeuge, die nicht um spezifische Kommandos und Darstellungen erweitert werden können.

1.5 Werkzeugbau mit *genform*

Ein Ergebnis der Arbeit ist der Werkzeug-Konstruktionsansatz *genform*. Mit *genform* können CASE-Werkbänke mit einer vollständigen Methodenunterstützung für die Analyse- und Entwurfsphase gebaut werden. *genform* enthält ein Framework mit Werkzeugkomponenten für CASE-Werkzeuge, mit denen die (graphischen) Benutzungsschnittstellen (*graphical user interface*, *GUI*), Werkzeugfunktionen und -kommandos, sowie die Datenverwaltung der Werkzeuge realisiert werden können [206, 246]. Der Werkzeugentwickler wird durch Meta-Werkzeuge beim Bau von CASE-Werkzeugen unterstützt. Die Meta-Werkzeuge sind auch mit *genform* gebaut.

1.5.1 Repository

Alle Daten einer CASE-Werkbank werden in einem zentralen *Repository* [33] verwaltet. Als Repository wird das objektorientierte Datenbank-Managementsystem H-PCTE [201] genutzt. Das Repository wird i.f. auch als *Object Management System*, kurz *OMS* , bezeichnet. Das Datenbankmodell von H-PCTE basiert auf dem ER-Modell: Es enthält Objekt- und Beziehungstypen, die beide Attribute besitzen können. Ein Objekttyp kann mehrere Elterntypen haben. Beziehungen werden in H-PCTE als *links* bezeichnet; eine H-PCTE-Datenbank auch als *Objektbank*.

Ein Datenbankschema definiert in H-PCTE eine Menge von Objekt- und Beziehungstypen mit ihren Attributen. Mit Datenbankschemata können *Sichten* auf die Objektbank definiert werden: Eine Sicht enthält eine Teilmenge der Typdefinitionen in der Objektbank und spiegelt die Sichtweise eines bestimmten Werkzeugs oder eines bestimmten Benutzers auf den Datenbestand wider. Typdefinitionen können zwischen verschiedenen Sichten importiert und somit wiederverwendet werden.

Das OMS übernimmt zahlreiche Aufgaben innerhalb einer CASE-Werkbank, dazu gehören [85] die Bereitstellung von Konzepten zur Modellierung von Software-Dokumenten und zur Definition verschiedener Sichten auf diese Dokumente, die persistente Verwaltung und Manipulation von Dokumenten, die Sicherung von Konsistenz und Integrität der Daten, sowie das Bereitstellen von Mechanismen für den Zugriffsschutz und die Synchronisation paralleler Zugriffe. Werden alle Daten im OMS verwaltet, dient es auch zur Datenintegration der verschiedenen Werkzeuge und ermöglicht die Kommunikation der Werkzeuge untereinander [319].

1.5.2 Werkzeugarchitektur

Ziel der verwendeten Werkzeugarchitektur ist es, möglichst viele der OMS-Dienste im Werkzeug zu nutzen. Dadurch wird der Implementierungsaufwand für die Werkzeuge verringert und dafür gesorgt, daß die verschiedenen Werkzeuge sich in ähnlichen Situationen gleich verhalten, etwa im Falle von Konsistenzverletzungen oder fehlenden Zugriffsrechten.

In herkömmlichen Werkzeugarchitekturen wird ein Dokument beim Werkzeugstart vom persistenten Speicher geladen und eine Kopie im Hauptspeicher angelegt [207]. Die Werkzeugkommandos arbeiten auf dieser Kopie des Dokuments. Ist die Bearbeitung beendet, wird die

Kopie auf den persistenten Speicher zurückgeschrieben. Hierbei ist es unerheblich, ob das Dokument im Dateisystem oder einem OMS abgelegt wird: Die im Werkzeug enthaltene *transiente Dokumentverwaltung* implementiert sowohl die Datenstrukturen für das Dokument, als auch die Operationen zu dessen Manipulation. Diese Werkzeugarchitektur hat folgende Nachteile:

- Der Aufwand für die Implementierung der transienten Dokumentverwaltung ist hoch und macht einen beträchtlichen Anteil am Implementierungsaufwand des Werkzeugs aus. Neben den Editieroperationen muß die Dokumentverwaltung auch Mechanismen für den Zugriffsschutz, die Konsistenzsicherung und Sichtendefinition enthalten.
- Wird ein OMS zur Speicherung der Dokumente verwendet, können seine Dienste zur Manipulation des Dokuments nicht genutzt werden. Sie kommen lediglich beim Laden und Speichern des Dokuments zum Einsatz.
- Änderungen am Dokument, die im Werkzeug durchgeführt werden, sind für andere Benutzer und Werkzeuge solange nicht sichtbar, bis das Dokument gespeichert wird. Dies ist problematisch, wenn mehrere Benutzer einen gemeinsamen Datenbestand bearbeiten – wie es bei der Softwareentwicklung häufig vorkommt.

Die von Kelter et al. vorgeschlagene *OMS-orientierte Werkzeugarchitektur* [208] löst die Probleme: Werkzeuge mit dieser Architektur arbeiten *direkt* auf der Objektbank, Editierkommandos des Werkzeugs werden also direkt auf OMS-Operationen abgebildet. Dadurch ist der Zustand der Objektbank stets aktuell und die Dienste des OMS können zur Implementierung der Werkzeugfunktionen genutzt werden – bis hin zu *Undo*- und *Redo*-Funktionen [270]. Der *Benachrichtigungsmechanismus* [271] von H-PCTE sorgt dafür, daß Werkzeuge über Änderungen in der Objektbank informiert werden.

Umgekehrt muß das Werkzeug auf Fehlermeldungen des OMS aufgrund von Konsistenzverletzungen oder fehlender Rechte geeignet reagieren oder vorher prüfen, ob ein Kommando beim aktuellen Zustand der Objektbank tatsächlich ausführbar ist. Ein Beispiel sind Zugriffsrechte: In einer herkömmlichen Werkzeugarchitektur müßte im Werkzeug auch eine Rechteverwaltung implementiert werden, die prüft, welche Operationen der Benutzer durchführen darf. In der OMS-orientierten Werkzeugarchitektur werden die Zugriffsrechte vom OMS verwaltet, das Werkzeug muß sich also nicht darum kümmern. Allerdings können Werkzeugkommandos aufgrund fehlender Rechte fehlschlagen. Das Werkzeug muß also entweder auf die Fehlermeldung geeignet reagieren oder besser nicht ausführbare Kommandos im voraus deaktivieren.

Folgende Voraussetzungen müssen erfüllt sein, damit die OMS-orientierte Werkzeugarchitektur funktioniert:

- Die Dokumente müssen *feingranular modelliert* sein, das OMS muß also die Feinstruktur der Dokumente kennen. Nur so können Werkzeugkommandos auf OMS-Operationen abgebildet und eine ausreichende Geschwindigkeit bei Editieroperationen erreicht werden. Ein Dokument wird daher im folgenden allgemein als Sammlung zusammenhängender Informationen betrachtet [286]. Aus Sicht des Werkzeugs kann auch eine einzelne Klasse oder eine Operation der Klasse als Dokument betrachtet werden.
- Das OMS muß eine ausreichende Geschwindigkeit bieten, damit die Antwortzeiten im Werkzeug nicht zu lang werden.

1.5.3 Schema-Interpretation

Sollen die Dokumente feingranular vom OMS verwaltet werden, so muß das Datenmodell der Dokumente in Datenbankschemata beschrieben werden. Andererseits muß das Datenmodell auch in den Werkzeugen bekannt sein: Sie müssen Kommandos zum Editieren von Attributen sowie zum Erzeugen und Löschen von Einträgen und Beziehungen anbieten. Entscheidend für die korrekte Funktion des Werkzeugs ist, daß das Datenbankschema im OMS und das Datenmodell im Werkzeug konsistent sind – und dies auch bei nachträglichen Änderungen bleiben.

Die von Däberitz et al. [69] vorgeschlagene *Schema-Interpretation* löst dieses Problem: Statt Datenmodelle im Datenbankschema und im Werkzeug redundant zu implementieren, ermittelt das Werkzeug Informationen über das Datenmodell direkt aus dem Datenbankschema:

- Anhand der Attribute von Objekt- und Beziehungstypen werden Formulare erzeugt, mit denen die Eigenschaften von Objekten und Beziehungen angezeigt und bearbeitet werden können.
- Anhand der Kompositionsbeziehungen zwischen Objekttypen können Komponenten eines Objekts dargestellt und Kommandos zum Erzeugen und Löschen von Komponenten angeboten werden.
- Anhand von Assoziationen zwischen Objekttypen können die ausgehenden Beziehungen eines Objekts angezeigt und manipuliert werden.

Die Interpretation des Datenbankschemas übernehmen *generische Werkzeuge*: Sie enthalten grundlegende Funktionen zur Darstellung und Manipulation von Dokumenten und werden mit dem Datenmodell eines konkreten Dokuments parametrisiert. H-PCTE bietet Operationen, mit denen Typeigenschaften von Objekten, Beziehungen und Attributen aus dem Schema ermittelt werden können. Das Datenbankschema, mit dem ein Werkzeug arbeitet, hängt von seiner aktuellen Sicht ab. Die Schema-Interpretation wird daher auch als *Sichtensteuerung* bezeichnet [68].

1.5.4 Werkzeugschemata

Natürlich reichen die Typeigenschaften im Datenbankschema nicht aus, um Aussehen und Verhalten eines konkreten Werkzeugs festzulegen. Zu den speziellen Werkzeugeigenschaften zählen die Darstellungen der Dokumente (etwa als Diagramm, Baum oder Tabelle) und die verfügbaren dokumentspezifischen Kommandos (etwa zum Prüfen und Transformieren von Dokumenten).

Die generischen Werkzeuge müssen also um werkzeugspezifische Funktionen erweitert werden:

- Die generischen Werkzeuge können neben dem Datenbankschema auch eine Beschreibung der Werkzeugeigenschaften interpretieren. Dabei muß die Konsistenz zwischen Datenmodell und Werkzeugeigenschaften gewahrt bleiben. Eine solche Spezifikation erleichtert den Werkzeugbau, hat aber auch eine eingeschränkte Flexibilität zur Folge.

– Die generischen Funktionen können in einem Framework implementiert werden. Die konkreten Werkzeuge nutzen diese generischen Komponenten und erweitern sie werkzeugspezifisch. Dies ist die flexibelste Möglichkeit, allerdings auch die aufwendigste und fehleranfälligste, da der Werkzeugentwickler das Framework kennen muß und keine Unterstützung für seine korrekte Verwendung erhält.

In *genform* werden beide Ansätze kombiniert: Einerseits wird das Datenmodell um eine Beschreibung der Werkzeugeigenschaften erweitert, die von den generischen Werkzeugkomponenten interpretiert wird; andererseits kann der Werkzeugentwickler Framework-Komponenten erweitern und in seine Werkzeuge einbinden. Die Werkzeugspezifikation, die Datenmodell und Werkzeugeigenschaften enthält, heißt *Werkzeugschema*.

In Abbildung 1.4 ist ein vereinfachtes Werkzeugschema für einen Klassendiagramm-Editor skizziert. Die Dokumentstruktur ist feingranular bis hin zu den Parametern einer Operation modelliert. Die Eigenschaften des Klassendiagramms, seiner Einträge und Beziehungen werden durch Attribute beschrieben. Zu beachten ist auch, daß das Datenmodell Eigenschaften enthält, die nur für einen graphischen Editor relevant sind (in der Abbildung kursiv dargestellt): Die Position einer Klasse im Diagramm oder die Stützpunkte einer Beziehung sind Informationen, die sich allein auf die graphische Darstellung beziehen. Die graphische Sicht kann also als Wiederverwendung und Erweiterung der Sicht eines textuellen Werkzeugs betrachtet werden.

Die Informationen zur Steuerung der Werkzeuge sind als grau hinterlegte Tabellen dargestellt und den Typen im Datenmodell zugeordnet.

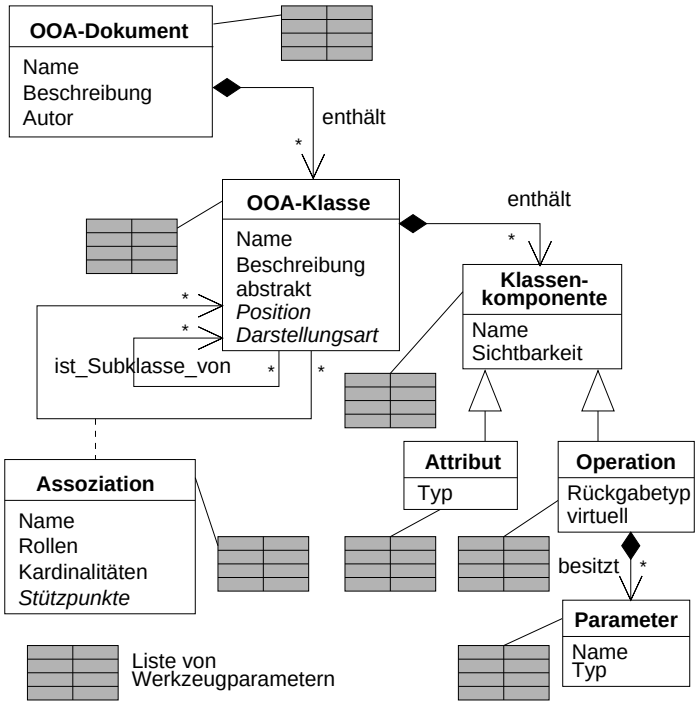


Abbildung 1.4: Vereinfachtes Werkzeugschema für einen Klassendiagramm-Editor

Werkzeugparameter

Werkzeugeigenschaften werden mit *Werkzeugparametern* spezifiziert. Werkzeugparameter sind den Typen im Datenmodell zugeordnet und können vom Werkzeug wie die Typeigenschaften ausgelesen werden. Technisch ist ein Werkzeugparameter ein Schlüssel-Wert-Paar. Es sind nur atomare Werte zulässig; mehrere Werkzeugparameter können zu Gruppen (ähnlich einem *record* in Pascal) zusammengefaßt werden. Mit Werkzeugparametern können Eigenschaften des GUI spezifiziert werden: Die Zeilenzahl eines Eingabefelds, die Benennung von Eingabefeldern oder das Aussehen eines Formulars. Werkzeugparameter können auch auf Typen aus dem Datenmodell verweisen, etwa wenn sie die Attribute festlegen, die in einer Tabelle oder einem Listeneintrag dargestellt werden sollen. Werkzeugparameter können auch die Zusammensetzung von Werkzeugen aus Komponenten steuern. Beispiele für den letzten Fall sind die Listenkomponente und die graphische Zeichenfläche – beide verwenden Komponenten zur Darstellung von Einträgen (und in der graphischen Zeichenfläche auch von Beziehungen), deren Typen mit Werkzeugparametern festgelegt werden können.

Die Zuordnung von Werkzeugparametern zu den Typen im Datenmodell hat folgende Vorteile:

- Die Konsistenz zwischen Werkzeugeigenschaften und dem Datenmodell kann leichter gewahrt werden.
- Werkzeugparameter von Objekttypen werden in der Typhierarchie vererbt, somit gemeinsame Eigenschaften in Obertypen zusammengefaßt.
- In PCTE können Typdefinitionen wiederverwendet werden, indem ein Schema Typen aus anderen Schemata *importiert*. Beim Import der Typdefinitionen werden auch die zugehörigen Werkzeugparameter importiert. Durch die Zuordnung von Datenbankschemata zu Werkzeugen werden somit Instanzen gleicher Typen in unterschiedlichen Werkzeugen auf gleiche Weise behandelt.

Das Konzept der (Datenbank-) Sicht wird somit auf die Werkzeugeigenschaften ausgedehnt: Ein Werkzeugschema definiert eine *Werkzeugsicht*. Unterschiedliche Werkzeugsichten können unterschiedliche Darstellungen für Dokumente definieren und unterschiedliche Kommandos enthalten. Beispiele für verschiedene Werkzeugsichten auf eine Klassenstruktur sind textuelle und graphische Editoren für Klassendiagramme und spezielle Editoren, die Fehler im objektorientierten Entwurf anzeigen und Kommandos zum Prüfen und Beheben der Fehler anbieten.

Ein weiterer Vorteil der Werkzeugparameter liegt in der Möglichkeit, Werkzeuge und Meta-Umgebung flexibel zu erweitern: Die Meta-Umgebung kann leicht erweitert werden, da den Typen im Datenmodell beliebige Werkzeugparameter zugeordnet und somit auch neue Eigenschaften ohne Änderungen an anderer Stelle spezifiziert werden können. Da die Werkzeugkomponenten die Werkzeugparameter interpretieren, ist in den Werkzeugkomponenten selbst festgelegt, wie sie auf definierte Parameter (oder das Fehlen derselben) reagieren. Es müssen also nur die Komponenten geändert werden, die einen neuen Parameter berücksichtigen sollen, es sind aber keine Änderungen in einer zentralen Komponente wie einem Compiler oder Generator nötig.

Andererseits ist die Erweiterung der Werkzeuge leicht möglich, weil die Werkzeuge aus beliebigen Werkzeugkomponenten zusammengesetzt werden können – somit auch aus Werkzeugkomponenten, die spezielle Werkzeugeigenschaften implementieren.

1.5.5 Werkzeugkomponenten

genform enthält ein Framework mit Werkzeugkomponenten zum Bau von CASE-Werkzeugen. Die enthaltenen Komponenten stellen also Funktionen zur Verfügung, die zur Implementierung einer großen Klasse von CASE-Werkzeugen gebraucht werden; andererseits sind die Funktionen genau auf diese Klasse von Werkzeugen beschränkt. Besondere Merkmale von *genform* sind:

- Die Werkzeugkomponenten sind *generisch*, sie interpretieren das Datenmodell und die Werkzeugparameter aus dem Werkzeugschema.
- Werkzeuge, die mit den Komponenten gebaut werden, arbeiten direkt auf der Objektbank, implementieren also die OMS-orientierte Werkzeugarchitektur.

Schichten

Die gebauten Werkzeuge weisen die (klassische) Dreischichten-Architektur auf:

1. *GUI*: Klassen für die Benutzungsschnittstelle implementieren Fenster, Interaktionselemente wie Eingabefelder, Formulare, Listen, Tabellen und Komponenten für graphische Editoren.
2. *Werkzeuglogik*: Die nächste Schicht enthält die Werkzeuglogik oder Werkzeugkommandos zum Erzeugen und Löschen von Einträgen und Beziehungen; zum Traversieren eines Dokuments und zum Durchführen von Prüfungen.
3. *OMS-Zugriffe*: In der untersten Schicht werden die Zugriffe auf die Objektbank durchgeführt. Sie enthält Komponenten, die das Erzeugen und Löschen von Objekten oder auch die Zwischenspeicherung von Daten aus dem OMS übernehmen. Letztere ist besonders wichtig, da die Benutzungsschnittstellen der Werkzeuge verteilt ausgeführt werden können und somit die Geschwindigkeit der Datenbankzugriffe gering ist. Diese Schicht enthält auch Komponenten, die Zugriffsrechte und die Sperrsituation an Ressourcen prüfen und im Werkzeug zur Aktivierung von Kommandos genutzt werden.

Die Klassen auf allen drei Ebenen sind generisch, enthalten also keine Informationen über das Datenmodell der Dokumente, mit denen sie arbeiten. Statt dessen lesen sie diese Informationen aus den Werkzeugschemata aus. Die Komponenten müssen also nicht auf der Ebene der Programmiersprache angepaßt werden, um mit einem gegebenen Datenmodell arbeiten zu können. Allerdings müssen einige grundsätzliche Annahmen über den Aufbau der Schemata vereinbart und in der Implementierung der Komponenten berücksichtigt werden.

Die OMS-orientierte Werkzeugarchitektur hat Auswirkungen auf alle drei Schichten: Aufgrund fehlender Rechte oder inkompatibler Sperren kann praktisch jedes Werkzeugkommando fehlschlagen. Im GUI muß daher die Situation im OMS vorher geprüft und Kommandos deaktiviert oder auf Fehlermeldungen passend reagiert werden. In der Implementierung der Werkzeugfunktionen müssen mögliche Fehler ebenfalls berücksichtigt werden: Sind Teile eines Dokuments nicht zugreifbar, muß die Prüfung des Dokuments abgebrochen oder das Prüfergebnis als unvollständig markiert werden. Die OMS-Zugriffsschicht übernimmt die Prüfung der Rechte- und Sperrsituation und bereitet die Informationen so auf, daß sie in den höheren Schichten leichter verarbeitet werden können.

Vergleich mit einer herkömmlichen Architektur

Im Vergleich zu einer herkömmlichen Werkzeugarchitektur ist die Koppelung zwischen den verschiedenen Schichten bei der OMS-orientierten Architektur höher, da die Funktionen der OMS-Zugriffsschicht praktisch von jedem Editierkommando verwendet werden, also nicht nur beim Laden und Speichern des Dokuments.

Die GUI-Schicht ist eher umfangreicher, da der Objektbankzustand sehr feingranular geprüft und die Interaktionsmöglichkeiten im GUI angepaßt werden müssen. Dies ist in herkömmlichen Werkzeugen meist nicht vorgesehen. Dafür fällt die Werkzeuglogik-Schicht bei der OMS-orientierten Architektur deutlich schmaler aus, da sie keine transiente Dokumentverwaltung enthält, sondern die Werkzeugkommandos direkt auf Funktionen der OMS-Zugriffsschicht abbildet.

Letztere ist umfangreicher als bei herkömmlichen Werkzeugen, da sie eine größere Zahl von OMS-Diensten den höheren Schichten zur Verfügung stellt, und den Zustand der Objektbank auswerten muß. Außerdem enthält sie die generischen Dokumentpuffer, die in einer herkömmlichen Architektur Teil der transienten Dokumentverwaltung sind.

Zusammenfassend hat die OMS-orientierte Architektur einen höheren Implementierungsaufwand in der GUI- und der OMS-Zugriffsschicht zur Folge, der durch den Verzicht auf die transiente Dokumentverwaltung kompensiert wird. Entscheidend ist, daß gemeinsame Funktionen an zentraler Stelle zusammengefaßt und von vielen Komponenten genutzt werden. Die Identifizierung dieser gemeinsamen Funktionen war bei der Entwicklung von *genform* ein iterativer Prozeß. Maßgeblich für die Bewertung ist aber der größere Funktionsumfang der Werkzeuge, den nachzubilden in herkömmlichen Werkzeugen einen deutlich größeren Aufwand erfordern würde.

1.5.6 Beispiel für eine CASE-Werkbank

In der Arbeit wird basierend auf *genform* die CASE-Werkbank *PI-SET* entwickelt. Dabei steht die Validierung des Werkzeugbaus mit *genform* im Vordergrund; nicht der Bau einer einsetzbaren Umgebung für einen bestimmten Anwendungsbereich. Der Prototyp einer Umgebung für den objektorientierten Entwurf wurde in einer Lehrveranstaltung validiert [248]. *PI-SET* enthält

- graphische und textuelle Editoren für verschiedene Dokumenttypen, darunter verschiedene Arten von UML-Diagrammen, ER-Diagramme und Petri-Netze.
- verschiedene Werkzeugsichten für unterschiedliche Aufgaben, z.B. graphische Editoren für Klassendiagramme in der Analyse-, Entwurfs- und Implementierungsphase [320].
- Funktionen zur Konsistenzsicherung und Transformation von Dokumenten. Letztere erzeugen Java-Quelltext aus einer Klassenstruktur oder Tabellendefinitionen aus einem ER-Diagramm.
- Planungs- und Managementwerkzeuge sowie eine Prozeßmaschine, mit denen Entwicklungsprozesse geplant und ausgeführt werden können (mehr dazu in Abschnitt 1.6.6).

1.5.7 Beitrag der Arbeit

Repository

Es wird praktisch gezeigt, wie CASE-Werkzeuge auf Basis eines objektorientierten Datenbank-Managementsystems realisiert werden können (s. dazu auch [34, 85, 104, 204, 235]): Die Software-Dokumente werden im Datenbankmodell des Repositories beschrieben und die Werkzeugfunktionen so direkt wie möglich auf die Dienste des OMS abgebildet. Dadurch wird der Implementierungsaufwand gegenüber herkömmlichen Werkzeugen mit gleichen Funktionen reduziert. Außerdem dient das Repository zur Datenintegration der verschiedenen Werkzeuge. Hierbei kommt dem Sichtenmechanismus des OMS eine große Bedeutung zu (*Multiple View Integration* [68]).

Werkzeugarchitektur

Die in der Arbeit beschriebenen Werkzeuge sind mehrbenutzerfähig und verteilt nutzbar. Die Werkzeugkomponenten müssen hierzu Zugriffsrechte und Sperren berücksichtigen und die Dienste des OMS für einen entfernten Zugriff nutzen.

Durch die Schichtenarchitektur des Frameworks können Werkzeugkomponenten einfacher wiederverwendet werden, da einzelne Komponenten sich auf jeweils einen bestimmten Aspekt konzentrieren; andererseits ist die Erweiterung und Wartung des Frameworks leichter möglich.

Beispiel 1.1 (Tabellendarstellung) In einer Tabellendarstellung (Abbildung 1.5) werden die Zielobjekte von ausgehenden Links eines Objekts dargestellt. Jedes Objekt erscheint als Tabellenzeile, jede Zelle enthält den Wert eines Attributs. In der Tabellendarstellung können Objekte über Knöpfe in einer Werkzeugleiste erzeugt und gelöscht werden; außerdem können die Attributwerte direkt editiert werden.

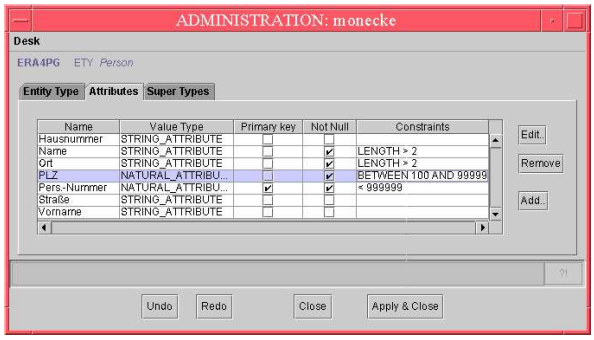


Abbildung 1.5: Werkzeug mit Tabellendarstellung

Geht man bei der Implementierung der Komponente von der Tabellenklasse eines GUI-Frameworks aus, muß diese Klasse so erweitert werden, daß Daten aus den OMS gelesen und im OMS manipuliert werden, daß die Kommandoknöpfe erzeugt werden und Änderungsoperationen wie das Erzeugen eines Objekts auslösen. Zusätzlich müssen Zugriffsrechte und Sperren berücksichtigt und Änderungsnachrichten verarbeitet werden. Die gleichen Funktionen müssen auch in einer Listendarstellung implementiert werden und sind ähnlich in einer Baum- oder graphischen Darstellung vorhanden.

Durch die Schichtenarchitektur beschränkt sich der Aufwand für die Tabellendarstellung auf die Implementierung der Darstellung der Daten in den Tabellenzellen – die Funktionen für den OMS-Zugriff, die Manipulation der Daten und die Erzeugung der Kommandoknöpfe kann aus der OMS- bzw. der Werkzeuglogik-Schicht wiederverwendet werden.

□

Für die Bearbeitung von *Mustern* im Datenmodell der Dokumente werden verschiedene GUI angeboten, die sich abhängig von den Anforderungen an das Werkzeug flexibel konfigurieren lassen. Beispiele für Muster sind Komponentenbeziehungen, die als Listen, Tabellen oder im Diagramm dargestellt werden können. Werkzeuge können einfach aus diesen GUI zusammengesetzt werden, so daß Prototypen mit geringem Aufwand gebaut werden können. Der Einsatz einer Teilmenge der Werkzeugkomponenten in textuellen Werkzeugen wird in [206] erläutert. Gleichzeitig können die Prototypen flexibel erweitert und dabei die Basisfunktionen des Frameworks genutzt werden.

Schemata

In der Arbeit wird die aus [68] bekannte Schema-Interpretation um *Typrechte* erweitert. Mit Typrechten können Zugriffsrechte auf Typebene spezifiziert werden: Den Objekt- und Linktypen sowie Attributen im Datenbankschema werden Zugriffsmodi zugeordnet. Diese legen fest, ob auf die Instanzen der Typen lesend und schreibend zugegriffen werden kann. Typrechte müssen also ähnlich wie Zugriffsrechte an Instanzen behandelt werden. Sie sind aber feingranularer, da auch die Manipulation von Attributen oder Beziehungen separat eingeschränkt werden kann.

Werkzeugschemata erweitern die Schema-Interpretation um die Möglichkeit, den Typen im Datenbankschema Werkzeugeigenschaften zuzuordnen. Eine CASE-Werkbank besteht aus mehreren Werkzeugen; die Spezifikation der Werkbank wird also auf mehrere Werkzeugschemata verteilt. In der Arbeit wird beschrieben, wie Beziehungen zwischen Werkzeugschemata definiert werden. Über diese Beziehungen kann der Benutzer einerseits innerhalb des Datenbestands navigieren; andererseits entsprechen sie Aufrufbeziehungen zwischen Werkzeugen, die durch Kommandos im GUI der Werkbank repräsentiert werden. Die Menge der verfügbaren Werkzeuge und die Eigenschaften der Werkzeuge sind jedoch nicht statisch, sondern sollen auf die Anforderungen des jeweiligen Benutzers und seiner Aufgabe abgestimmt sein. In der Arbeit wird vorgeschlagen, wie die Werkbank sich automatisch an die aktuelle Rolle und Aufgabe des Benutzers und den aktuellen Zustand des Entwicklungsprozesses anpaßt.

1.6 Methodenunterstützung

Mit *genform* können Werkzeuge gebaut werden, die an die Konzepte und Notationen einer gegebenen Methode angepaßt sind. Um eine vollständige Methodenunterstützung zu realisieren, muß aber auch das Prozeßmodell der Methode berücksichtigt werden. Das Prozeßmodell kann den Entwicklern als Anleitung bei der Prozeßausführung dienen, und ihnen können die jeweils passenden Werkzeuge zur Ausführung einer Aufgabe angeboten werden. Letzteres erfordert die Kombination von Prozeßmodell und Werkzeugspezifikationen sowie die Verwaltung des aktuellen Prozeßzustands während der Prozeßdurchführung.

Eine prozeßorientierte Software-Entwicklungsumgebungen enthält neben den CASE-Werkzeugen auch Werkzeuge, die die Planung und Ausführung von Prozessen unterstützen. In der Arbeit wird *genform* zum Bau solcher PSEU genutzt und dabei geprüft, wie die vorhandenen Konzepte zum Werkzeugbau beim Bau von PSEU eingesetzt werden können und welche Erweiterungen nötig sind.

Das OMS verwaltet in einer PSEU neben den Dokumenten auch den Prozeßzustand. Es werden also Datenbankschemata für diese Daten definiert, Beziehungen zwischen den Dokumentdatenmodellen und dem Datenmodell für den Prozeß identifiziert und es wird geprüft, wie weit die Dienste des OMS auch zur Manipulation des Prozeßzustands eingesetzt werden können.

Das Vorgehensmodell besteht aus Arbeitsschritten, die ausgeführt werden müssen, um das angestrebte Ziel zu erreichen – hier die Produktion einer konsistenten Beschreibung des zu realisierenden Systems. Ein Arbeitsschritt kann auch als Spezifikation einer Aufgabe aufgefaßt werden, die von einer Person ausgeführt werden muß. Diese Spezifikation enthält:

- das Ziel der Aufgabe
- die möglichen Zustände der Aufgabe
- Anforderungen an den Bearbeiter (ausgedrückt durch seine Rolle)
- die Typen der Dokumente, die beim Ausführen der Aufgabe gelesen, erzeugt oder verändert werden
- die Werkzeuge, die in der Aufgabe verwendet werden

Weiterhin müssen Beziehungen zwischen Aufgaben festgelegt werden. Diese wirken sich auf die möglichen Reihenfolgen aus, in denen Aufgaben ausgeführt werden können, und bestimmen die Zerlegung von Aufgaben in Teilaufgaben (s. Abschnitt 1.6.1).

1.6.1 Eine einfache Prozeßmodellierungssprache

Zur Beschreibung von Prozessen wird eine einfache graphische Prozeßmodellierungssprache vorgestellt. Diese dient allerdings nur zur Demonstration der entwickelten Konzepte und ist nicht eigentlicher Forschungsgegenstand. Es wurde keine der zahlreichen bekannten Prozeßmodellierungssprachen [62] verwendet, weil

- diese Sprachen meist sehr komplex sind, also schwierig zu erlernen und zu verwenden. Für Experimentierzwecke reicht aber einer einfache Sprache aus, auch wenn mit ihr nicht alle Details des Prozesses beschrieben werden können.
- die zugehörigen Laufzeitsysteme wie Interpreter oder Generatoren komplex sind. Ein solcher Interpreter hätte entweder wiederverwendet und mit *genform* integriert oder nachimplementiert werden müssen. Beides erfordert einen großen Aufwand und umfangreiches Wissen über das wiederverwendete System.
- das Prozeßmodell und die resultierenden Instanzen im OMS verwaltet und mit *genform*-Werkzeugen bearbeitet werden sollen. Dadurch wird die Flexibilität von *genform* validiert, und geprüft, wie die OMS-Dienste zur Verwaltung des Prozeßzustandes ausgenutzt werden können. Schließlich wird auch die enge Prozeßintegration der CASE-Werkzeuge durch die gemeinsame Verwaltung von Software-Dokumenten und Prozeßzustand im OMS erreicht.

Abbildung 1.6 zeigt ein Beispiel für die graphische Darstellung eines Prozeßmodells in der vorgeschlagenen Notation:

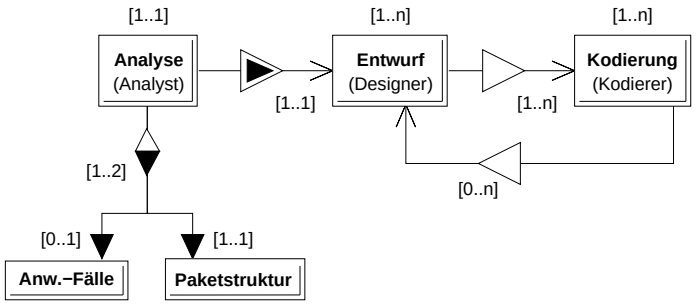


Abbildung 1.6: Beispiel für ein Prozeßmodell

Die graphische Darstellung enthält *Aufgabentypen* und die Beziehungen zwischen Aufgabentypen: Nachfolger-Beziehungen beeinflussen die Reihenfolge, in der Aufgaben ausgeführt werden können (erst Analyse, dann Entwurf); über Teil-von-Beziehungen können Aufgaben in Teilaufgaben zerlegt werden (die Analyse setzt sich zusammen aus dem Aufstellen von Anwendungsfällen und dem Erstellen der Paketstruktur). Das graphische Modell beschreibt also die Struktur oder die *Architektur* eines Prozesses [117]. Über *Kardinalitäten* an Aufgabentypen und Beziehungen kann die Anzahl der möglichen Instanzen eingeschränkt werden. Im Aufgabentyp kann auch die erforderliche *Rolle* eingetragen werden.

Das Prozeßmodell wird als Datenmodell für Prozeßinstanzen aufgefaßt. Um die Prozeßinstanzen im OMS verwalten zu können, wird aus dem Datenmodell ein Datenbankschema erzeugt. Dies ähnelt dem Vorgehen in *DYNAMITE* [157], wo aber neben der Struktur des Prozesses auch die möglichen Änderungsoperationen in Graphschemata beschrieben werden. Die Vorteile des gewählten Ansatzes sind:

- Die Dienste des OMS zur Schemaverwaltung und zur Konsistenzsicherung werden zur Verwaltung des Prozeßmodells und zur Sicherung der Konsistenz der Prozeßinstanz genutzt – die Dienste müssen also nicht von der Umgebung implementiert werden.
- Die Steuerung der Werkzeuge durch das Datenbankschema funktioniert auch für die Prozeßinstanz. Die Werkzeuge zur Darstellung und Manipulation des Aufgabennetzes können also einfach gebaut werden.

Nachteilig ist, daß bei der Administration der Prozeßmodelle auch Datenbankschemata erzeugt und geändert werden müssen: Das Generieren der Datenbankschemata aus dem Prozeßmodell und das Einrichten der Schemata im OMS ist aufwendig und erfordert Administrationsrechte beim Prozeßmodellierer. Bei einer nachträglichen Änderung des Datenbankschemas müssen auch vorhandene Instanzen konvertiert werden, was nicht alle DBMS unterstützen. So gehört die Änderbarkeit der Datenbankschemata auch zu den Anforderungen an OMS für Software-Entwicklungsumgebungen [294]. PCTE unterstützt die Schemaevolution und die Konvertierung der Instanzen.

Problematischer ist die Evolution des Prozeßmodells und die Anpassung der vorhandenen Prozeßinstanzen [29], da ein laufender Prozeß nicht einfach in einen anderen 'konvertiert'

werden kann. Probleme treten auf, wenn ein Aufgabentyp gelöscht wird, von dem noch aktive Instanzen existieren, oder wenn die Vorbedingung einer Aufgabe nicht mehr erfüllt werden kann, weil der Vorgänger nicht mehr existiert. Diese Fragen werden hier nicht behandelt. Statt dessen gehen wir davon aus, daß ein Umgebungsadministrator mit passenden Rechten die Umgebung und auch die Datenbankschemata einmalig einrichtet.

Neben den Aufgaben und ihren Abhängigkeiten muß auch beschrieben werden, welche Arten von Dokumenten in den einzelnen Aufgaben bearbeitet und auf welchem Weg sie im Prozeß weitergeleitet werden. In Abbildung 1.7 ist dieser Aspekt des Prozesses dargestellt: Relevante

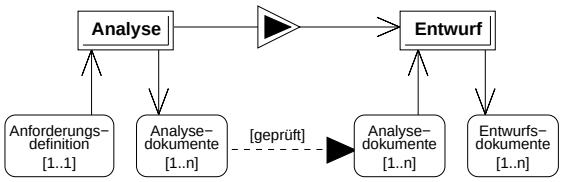


Abbildung 1.7: Dokumentordner mit Datenflußbeziehung

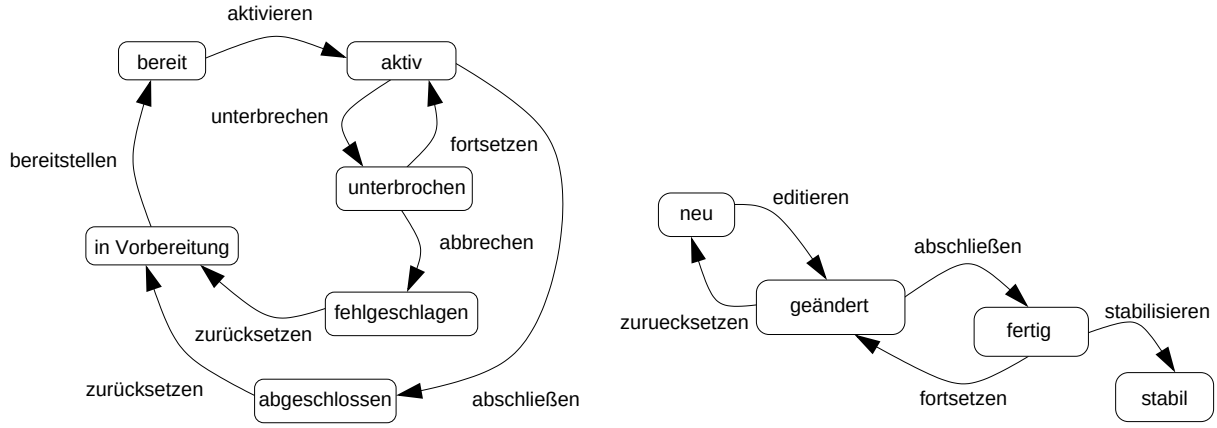
Dokumente einer Aufgabe werden in *Dokumentordnern* verwaltet. Dokumente können zwischen den Ordnern über *Datenflußbeziehungen* weitergeleitet werden.

Weitere Eigenschaften des Prozesses sind in der graphischen Darstellung nicht sichtbar, aber zur Ausführung des Prozesses wichtig:

- Die möglichen Zustände von Aufgaben und Dokumenten werden in *Zustandsdiagrammen* definiert. An den Zustandsübergängen werden Bedingungen und Aktionen notiert, die bestimmen, wann ein Zustandswechsel erlaubt ist bzw. was beim Zustandswechsel zu tun ist. Bei Aufgaben wird das Zustandsdiagramm dem Aufgabentyp zugeordnet. Die Zustände der Dokumente werden aufgabenspezifisch definiert: Dokumente eines Typs besitzen in unterschiedlichen Aufgaben unterschiedliche Zustandsdiagramme. Genauer werden die Zustandsdiagramme den Dokumentordnern zugeordnet, in denen sich die Dokumente einer Aufgabe befinden. In Abbildung 1.8 sind die vorgegebenen Zustandsdiagramme für Aufgaben und Dokumente skizziert. Um den Prozeßablauf zu beeinflussen, kann der Prozeßmodellierer den Zustandsübergängen Bedingungen und Aktionen zuordnen oder die Zustandsdiagramme selbst modifizieren. Aufgabenspezifische Zustandsdiagramme werden auch in *APEL* [72] verwendet. Mehr zur Implementierung der Zustandsautomaten in Abschnitt 1.6.5.
- Durch die Zuordnung von CASE-Werkzeugen zu Aufgaben wird die Bearbeitung der Aufgaben erleichtert. Gleichzeitig kann gewährleistet werden, daß die richtigen Werkzeuge genutzt werden, mit den vorgesehenen Darstellungen und Kommandos. Genauer werden den Dokumentordnern Werkzeugschemata zugeordnet. Somit können die Eigenschaften der Werkzeuge, die zur Bearbeitung der Dokumente im Ordner genutzt werden sollen, sehr genau bestimmt werden.

Prozeßinstanz

Die Instanz eines Prozesses kann als *Aufgabennetz* aufgefaßt werden: Die Knoten des Netzes sind die Aufgaben mit ihren unterschiedlichen Zuständen, die Verbindungen zwischen Aufgaben werden durch Nachfolger- und Teilaufgabenbeziehungen gebildet.



(a) Aufgaben (b) Dokumente

Abbildung 1.8: Vorgegebene Zustandsdiagramme

Aus dem Prozeßmodell wird das Datenbankschema abgeleitet, mit dem Aufgabennetze in der Objektbank verwaltet werden können. Aufgaben werden in der Objektbank durch Objekte, Beziehungen zwischen Aufgaben durch Links repräsentiert (Abbildung 1.9). Damit

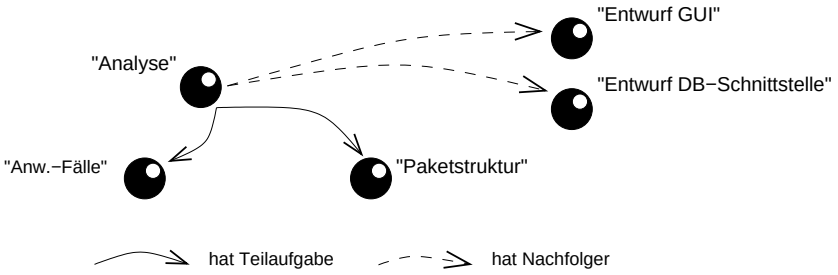


Abbildung 1.9: Prozeßinstanz in der Objektbank

der Prozeß in einer PSEU tatsächlich ausgeführt werden kann, müssen neben den Datenbankschemata auch Benutzer und Benutzergruppen in der Objektbank eingerichtet werden. Letztere repräsentieren die Rollen im Prozeß. Außerdem müssen Datenstrukturen initialisiert werden, die die Prozeßmaschine zur Verwaltung des Prozeßzustands benötigt.

1.6.2 Kombination von Werkzeugspezifikation und Prozeßmodell

Aus Benutzersicht sind die Dokumentordner Arbeitsmappen, die die benötigten Dokumente für eine Aufgabe enthalten. Im Modell werden die Typen von Dokumentordnern den Aufgabentypen zugeordnet (Abbildung 1.10).

Der Typ eines Dokumentordners definiert

– den Typ der enthaltenen Dokumente. Ein Dokumentordner verwaltet Dokumente von genau einem Typ.

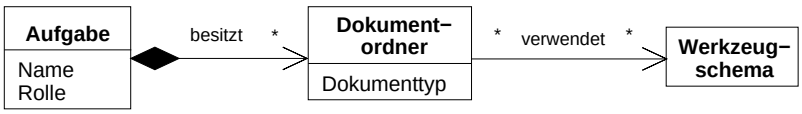


Abbildung 1.10: Ausschnitt aus dem Metamodell für Dokumentordner

- die Zugriffsrechte an den enthaltenen Dokumenten. Der Bearbeiter der Aufgabe muß Schreib- oder Leserechte an den enthaltenen Dokumenten besitzen; zusätzlich können Rechte für andere Benutzer festgelegt werden. Dadurch wird die gemeinsame Bearbeitung eines Dokuments ermöglicht.
- die Werkzeugsichten, mit denen die Dokumente bearbeitet werden. Die Werkzeugsichten werden durch Werkzeugschemata beschrieben.

Die Dokumentordner legen somit den *Arbeitsbereich* (s. z.B. [323]) mit den benötigten Dokumenten und Werkzeugen fest. Die Werkzeuge und ihre Varianten werden im Prozeßmodell verwaltet.

1.6.3 Integration von CASE-Werkzeugen und Prozeß

Der aktuelle Zustand des Entwicklungsprozesses setzt sich zusammen aus den Zuständen von Aufgaben und den Zuständen der bearbeiteten Dokumente. Abhängig vom Prozeßzustand sind nur bestimmte Tätigkeiten sinnvoll: Mit dem Entwurf eines Teilsystems kann erst begonnen werden, wenn die zugehörige Analyse abgeschlossen ist; die Dokumentation sollte erst generiert werden, wenn sich der Entwurf in einem konsistenten Zustand befindet, und ein Dokument sollte erst geändert werden können, nachdem es einem Entwickler zur Bearbeitung überlassen wurde.

Beeinflussung des Prozeßzustands

Die möglichen Zustände von Aufgaben und Dokumenten werden im Prozeßmodell beschrieben. Ein Entwickler, der im Prozeß arbeitet, kann den Prozeßzustand auf folgende Arten ändern:

- Indem er den Zustand einer Aufgabe ändert, also etwa eine Review-Aufgabe als abgeschlossen markiert.
- Indem er den Zustand eines Dokuments ändert. Sind im Prozeßmodell die Zustände 'in Bearbeitung' und 'fertig' für ein Entwurfsdokument definiert, so stellt der Entwickler durch die Änderung des Zustands auf 'fertig' das Dokument für folgende Arbeitsschritte zur Verfügung.
- Indem er Dokumente bearbeitet. Beim Editieren eines Dokuments wird implizit sein Zustand geändert, etwa beim Beseitigen von Fehlern. Die impliziten Zustände müssen auf die im Prozeßmodell definierten Zustände abgebildet werden. Dies übernehmen zum Beispiel Prüfwerkzeuge, die ein Dokument untersuchen und abhängig vom Ergebnis der Prüfung den expliziten Dokumentzustand ändern.

Der Zustand der bearbeiteten Dokumente hat Einfluß auf den Zustand der Aufgabe: Befinden sich alle zu bearbeitenden Entwurfsdiagramme im Zustand 'fertig', so ist auch die zugehörige Entwurfsaufgabe abgeschlossen. Zur Abbildung von Dokumentzuständen auf die Zustände der Aufgaben muß die Prozeßmaschine auf die Dokumente zugreifen und Prüfkommandos ausführen können.

In PSEU, die mit *genform* gebaut werden, ist eine solche Integration leicht möglich, weil die Dokumente und der Prozeßzustand im gemeinsamen Repository verwaltet werden, und die Prozeßmaschine die Datenmodelle der Dokumente (genauer die Werkzeugschemata) kennt. Dank der feingranularen Modellierung kann die Prozeßmaschine Analysen auf den Dokumentinhalten ausführen und hierzu auch Komponenten von Prüfwerkzeugen nutzen. Prozeßmaschine und CASE-Werkzeuge arbeiten also auf den gleichen Daten und können gemeinsame Komponenten verwenden, dadurch wird die Integration von Prozeßmaschine und Werkzeugen erleichtert und der Implementierungsaufwand für die PSEU verringert.

Umgekehrt können die Werkzeuge auch auf Informationen des Prozesses zugreifen: Im CASE-Werkzeug kann ein Entwickler den Zustand eines Dokuments auch explizit ändern. Dies funktioniert, weil das CASE-Werkzeug den Teil des Prozeß-Datenmodells kennt, in dem Dokumentzustände verwaltet werden – das Werkzeugschema importiert dazu Typen aus dem Prozeß-Schema. Das Werkzeug enthält aber keine speziellen Komponenten für die Interaktion mit dem Prozeß. Somit wird die Prozeßintegration der Werkzeuge allein über die Werkzeugschemata realisiert und erfordert keine Anpassung der Werkzeugkomponenten.

Anpassung der Werkzeuge an den Prozeßzustand

In *genform* beschränkt sich die Anpassung der Werkzeuge auf Modellebene daher auf die Definition von Werkzeugschemata und die Zuordnung der Werkzeugschemata zu Aufgaben im Prozeßmodell. Ein Werkzeugschema entspricht einer 'Konfiguration' von Werkzeugeigenschaften; Teile dieser Konfigurationen können in mehreren Werkzeugschemata wiederverwendet werden.

Bei der Anpassung der Werkzeuge an den Prozeßzustand wird ausgenutzt, daß die Werkzeuge direkt auf der Objektbank arbeiten und die Dokumente feingranular modelliert sind: Durch das Setzen von Zugriffsrechten an den Dokumenteinträgen können die im Werkzeug angezeigten Daten und die durchführbaren Kommandos beeinflußt werden. Der Entwickler muß keine weiteren Werkzeuge oder Benutzungsschnittstellen bedienen und keine zusätzlichen Arbeitsschritte ausführen, um mit dem Prozeß zu interagieren. Außerdem müssen Daten wie Dokumentlisten oder Dokumentzustände nicht redundant im Datenmodell der Werkzeuge und im Prozeßdatenmodell verwaltet werden.

In *PRIME* [274] werden Werkzeuge im Modell an die unterschiedlichen Prozeßsituationen angepaßt: Die verschiedenen Situationen werden dort als *Kontexte* definiert. Ein Kontext legt fest, welche Kommandos im Werkzeug ausgeführt werden können. Auf diese Weise können Werkzeuge sehr feingranular angepaßt werden, bis hin zum Verhalten des GUI beim Erzeugen einer Verbindung zwischen zwei Dokumenteinträgen. Allerdings ist dieser Ansatz sehr aufwendig, da die möglichen Prozeßsituationen im voraus beschrieben und ihre Auswirkungen auf die Werkzeuge festgelegt werden müssen. Außerdem scheint die angestrebte Unterstützung eher für ungeübte Benutzer sinnvoll, die eine sehr ausführliche Anleitung auf

Basis sehr kleiner Arbeitsschritte benötigen.

1.6.4 Prozeßwerkzeuge

Neben den CASE-Werkzeugen müssen in einer PSEU auch Werkzeuge vorhanden sein, über die Benutzer mit dem Prozeß interagieren können:

Mit Planungswerkzeugen wird der Projektverlauf im voraus geplant. Die Planungswerkzeuge sollten das Prozeßmodell als Grundlage für die Planung verwenden, also nur die definierten Aufgabentypen zulassen und die Rolle der Bearbeiter und die Abhängigkeiten zwischen Aufgaben berücksichtigen. Die Planungssicht enthält zusätzlich Informationen über Aufwände, Kosten und Termine.

Während der Ausführung des Prozesses sind zwei Sichten auf den Prozeßzustand wichtig:

- Ein Manager ist eher daran interessiert, einen Überblick über die aktuelle Situation zu erhalten [323], um den Zustand des Prozesses beurteilen und mit dem Plan vergleichen zu können.
- In der Sicht des Entwicklers stehen seine eigenen Aufgaben im Vordergrund: Eine *Agenda* zeigt seine Aufgaben an, zusammen mit Informationen wie dem aktuellen Zustand und dem Fertigstellungstermin. Die Agenda kann auch als Ausgangspunkt für die Ausführung der Aufgaben genutzt werden, wenn sie die Navigation zu den Dokumenten und das Starten von Werkzeugen ermöglicht.

Beim Bau der Prozeßwerkzeuge müssen zwei Anforderungen berücksichtigt werden: Zum einen müssen die Werkzeuge möglichst gut in die PSEU integriert sein, also eine ähnliche Benutzungsschnittstelle aufweisen und Daten mit den CASE-Werkzeugen austauschen können. Dadurch wird die Akzeptanz der Werkzeuge beim Benutzer erhöht und der Aufwand für ihre Nutzung verringert.

Zum anderen muß der Bau und die Integration der Werkzeuge mit geringem Aufwand möglich sein, um die Kosten für die PSEU zu begrenzen. Ziel ist es also, die Gemeinsamkeiten zwischen CASE- und Prozeßwerkzeugen in gemeinsamen Komponenten zusammenzufassen.

Bau von Prozeßwerkzeugen mit *genform*

Die Werkzeuge für verschiedene Prozeßmodelle unterscheiden sich, da sie die unterschiedlichen Eigenschaften der Prozeßmodelle widerspiegeln. Solche Eigenschaften sind:

- Die enthaltenen Aufgabentypen und ihre jeweiligen Eigenschaften: Aufgabentypen können Informationen wie aktuelle Aufwände oder die Zahl der zu behebenden Fehler zugeordnet werden. Zum Erzeugen von Aufgaben und zum Bearbeiten der Eigenschaften müssen passende GUI verfügbar sein.
- Die möglichen Zustände einer Aufgabe hängen vom Aufgabentyp ab und müssen über ein GUI änderbar sein.
- Dokumente und Bearbeiter müssen den Aufgaben über ein GUI zugeordnet werden können. Die Typen der Dokumente und die Rollen der Bearbeiter hängen vom Prozeßmodell ab.

– Erlaubte Beziehungen zwischen Aufgaben wie Nachfolger- und Teilaufgaben-Beziehungen sind im Prozeßmodell definiert. Im GUI müssen die Konsistenzbedingungen des Prozeßmodells berücksichtigt werden.

In *LEU* wird die Benutzungsschnittstelle von Prozeßwerkzeugen aus dem zugehörigen Datenbankschema generiert [82]. Mit den Werkzeugen können auch die Eigenschaften von Prozeßelementen bearbeitet werden – in einem Geschäftsprozeß zur Immobilienverwaltung etwa Mieter oder Wohnungen. *ADDD* [213] enthält allgemeine GUI; prozeßspezifische können mit beliebigen Werkzeugen gebaut und über ein API eingebunden werden.

Soll ein Prozeßwerkzeug mit *genform* gebaut werden, muß zunächst ein Werkzeugschema erzeugt werden. Das Werkzeugschema kann wiederum aus dem Prozeßmodell abgeleitet werden: Aus den Aufgabentypen mit ihren Eigenschaften und Beziehungen wird das Datenmodell der Werkzeuge erzeugt und um Werkzeugparameter ergänzt. Die Werkzeugschemata können dann erweitert werden, etwa, um die (graphische) Darstellung von Prozeßelementen zu beeinflussen oder Kommandos hinzuzufügen. Die Werkzeugschemata der verschiedenen Prozeßwerkzeuge werden durch Erweiterung dieses Basis-Werkzeugschemas erzeugt (Abbildung 1.11).

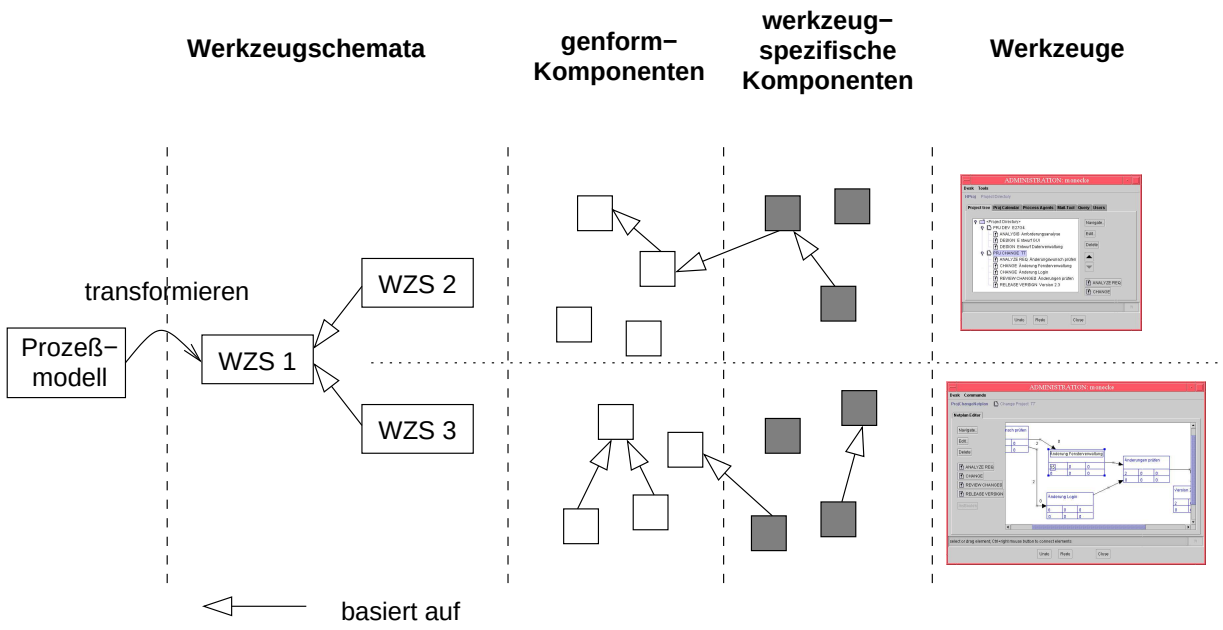


Abbildung 1.11: Generieren von Prozeßwerkzeugen

Die Werkzeugschemata enthalten das Datenmodell des jeweiligen Werkzeugs zusammen mit den Werkzeugparametern, die die Werkzeugeigenschaften festlegen. Zusätzlich müssen werkzeugspezifische Komponenten implementiert werden, etwa für die graphische Darstellung des Plans und des Prozeßzustands, für die Kalenderdarstellung oder die Netzplanberechnung [289].

1.6.5 Prozeßmaschine

Die Prozeßmaschine überwacht Änderungen im Prozeß und reagiert darauf, indem sie selbst den Prozeßzustand und damit den Fortschritt des Prozesses beeinflußt. In *genform* ist jeder PSEU-Instanz eine eigene Instanz der Prozeßmaschine zugeordnet. Da der Prozeßzustand in der Objektbank verwaltet wird, arbeitet auch die Prozeßmaschine auf dem Repository und kommuniziert mit den Werkzeugen der PSEU über das Repository.

Der Vorteil dieses Ansatzes ist, daß keine Kommunikationskomponente wie ein *broadcast message server* [280] benötigt wird. Auch muß die Kommunikation nicht explizit in den Werkzeugen implementiert werden: Sie reagieren automatisch auf den Zustand der Objektbank.

Zu den Aufgaben der Prozeßmaschine gehören

- das Erzeugen von Aufgaben und Beziehungen zwischen Aufgaben.
- die Verwaltung von Aufgaben- und Dokumentzuständen.
- das Weiterleiten von Dokumenten zwischen Aufgaben.
- das Ausführen von Aktionen infolge von Ereignissen im Prozeß. Beispiele sind das Versenden von Nachrichten an die Benutzer oder das Erzeugen eines abgeleiteten Dokuments.

Aufbau der Prozeßmaschine

In der Architektur der Prozeßmaschine können drei Schichten unterschieden werden:

1. *Prozeßmodell*: Die oberste Schicht implementiert die Logik des Prozesses. Diese Schicht kann Komponenten mit prozeßspezifischen Funktionen enthalten oder aus einem Interpreter bestehen, der eine Prozeßbeschreibung ausführt.
2. *Manipulation des Prozeßzustands*: Die zweite Schicht enthält Funktionen zur Überwachung und Änderung des Prozeßzustands. Sie hängt von der konkreten Repräsentation des Prozeßzustands im Repository ab.
3. *OMS-Zugriff*: Die unterste Schicht implementiert die OMS-Zugriffsoperationen und legt fest, wie die Dienste und Mechanismen des OMS genutzt werden.

Die Komponenten der OMS-Zugriffsschicht sind bereits in *genform* implementiert und werden in den CASE-Werkzeugen verwendet, so daß hier kein zusätzlicher Implementierungsaufwand entsteht. Die nächsthöhere Schicht abstrahiert von der konkreten Speicherung des Prozeßzustands in der Objektbank. Hier werden Funktionen angeboten zur Manipulation von Aufgaben, zum Zuordnen von Dokumenten zu Aufgaben und zur Verwaltung von Zuständen. Wie diese Funktionen auf die OMS-Operationen abgebildet werden, hängt vom Datenmodell ab, in dem Prozesse verwaltet werden.

Die oberste Schicht enthält prozeßspezifische Funktionen. Hier ist festgelegt, welche Änderungen des Prozeßzustands erlaubt sind, und was als Reaktion auf diese Änderungen passiert. In Ansätzen zur *Prozeßprogrammierung* wie *APPL/A* [299] wird das Prozeßmodell als Programm aufgefaßt und in einer Programmiersprache (dort eine Erweiterung von Ada) implementiert. Andere Ansätze benutzen graphische Sprachen wie *FUNSOFT-Netze* [141]

oder *SLANG* [19], um den Prozeß zu beschreiben. Das Netz wird dann kopiert und mit einem Zustand versehen (*active copy*), der während der Prozeßausführung gemäß den im Netz definierten Transitionen verändert wird.

In *genform* werden beide Ansätze kombiniert: Die Struktur des Prozesses wird mit der graphischen Prozeßmodellierungssprache aus Abschnitt 1.6.1 beschrieben und das Prozeßmodell von der Prozeßmaschine interpretiert. Die Aktionen, die im Prozeß ausgeführt werden sollen, sind als Komponenten der Prozeßmaschine implementiert. Hierfür sind Basiskomponenten und Abstraktionen in *genform* enthalten, die prozeßspezifisch erweitert werden können.

Ähnlich wie beim Werkzeugbau kann also auch die Prozeßmaschine durch eine Spezifikation (das Prozeßmodell) adaptiert und können spezielle Anforderungen durch prozeßspezifische Komponenten berücksichtigt werden.

Prozeßagenten

Zur Laufzeit des Prozesses enthält die Prozeßmaschine eine Hierarchie von *Prozeßagenten*. Prozeßagenten sind die aktiven Einheiten in der Prozeßmaschine: Sie überwachen Änderungen des Prozeßzustands und beeinflussen ihrerseits den weiteren Prozeßfortschritt. Beim Bau der Prozeßmaschine sind die Agenten wiederverwendbare Bausteine, anpaßbar an die Anforderungen des Prozesses.

Es existieren zahlreiche Ansätze, die das Konzept des Agenten zur Prozeßausführung nutzen. Ansätze wie *ADEPT* [184], *Redux* [269] oder *PEACE+* [7] verwenden *rollenbasierte autonome Agenten*: Jeder Agent verfügt über ein bestimmtes Wissen und versucht, damit seine Aufgabe zu erfüllen. Bei Konflikten verhandeln die Agenten miteinander, um eine einvernehmliche Lösung zu finden. Die Struktur des Prozesses ergibt sich also aus der Organisationsstruktur der kooperierenden Agenten.

Bei *aufgabenbasierten Ansätzen* wird zunächst der Prozeß explizit beschrieben. Den einzelnen Aufgaben im Prozeß werden Agenten zugeordnet, die für die Ausführung der Aufgaben zuständig sind, und dabei das Prozeßmodell interpretieren. Beispiele sind *SCALE* [266] und *MOKASSIN* [136].

Auch in *genform* ist eine Agentenhierarchie jeweils einer Aufgabe zugeordnet. Genauer besitzt jede Aufgabe einen *Task Execution Agent* (TEA), der für die Ausführung der Aufgabe zuständig ist (Abbildung 1.12). Der TEA lädt weitere Agenten, so daß eine Hierarchie entsteht. Das Verhalten der Agenten wird zum einen durch das Prozeßmodell beeinflußt, zum anderen können die Agenten auch prozeßmodellspezifisch erweitert werden. Somit beschreibt das graphische Prozeßmodell die Architektur des Prozesses, während seine Implementierung in den Prozeßagenten steckt. Abbildung 1.13 verdeutlicht den Zusammenhang zwischen der Repräsentation des Prozeßzustands in der Objektbank und den Prozeßagenten in der Prozeßmaschine.

Zustandsautomaten

Am Beispiel der Zustandsautomaten für Aufgaben und Dokumente soll die Erweiterung und Anpassung der Prozeßmaschine näher erläutert werden: Im Prozeßmodell werden die Zustände durch Zustandsdiagramme beschrieben. Der Prozeßmodellierer bearbeitet die Zustandsdiagramme mit einem graphischen Editor. Bedingungen und Aktionen werden an den

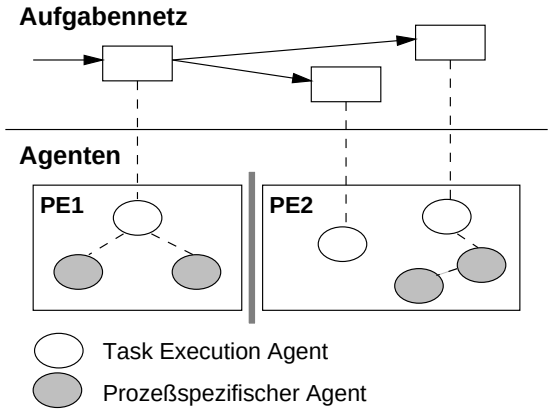


Abbildung 1.12: Aufgaben und Agenten

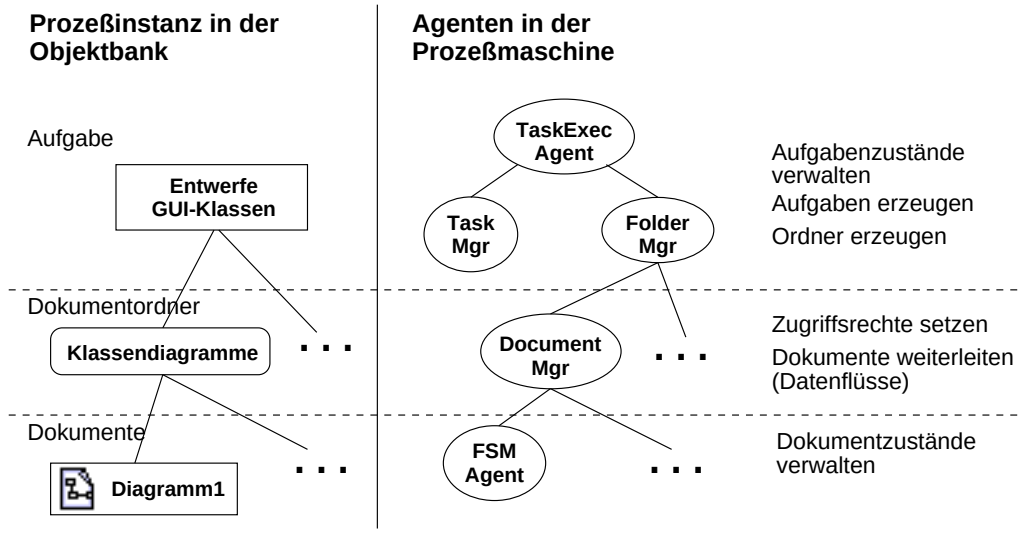


Abbildung 1.13: Agenten in der Prozeßmaschine

Zustandsübergängen notiert. Eine Bedingung kann aus der logischen Verknüpfung mehrerer Ausdrücke bestehen; Aktionen werden als Folge von Kommandos definiert und mit Parametern versehen.

Bei der Ausführung des Prozesses werden die Zustandsdiagramme von der Prozeßmaschine interpretiert. Ein Zustandswechsel kann auf verschiedene Arten ausgelöst werden: Auf Anforderung des Benutzers, durch ein Werkzeug oder automatisch durch die Prozeßmaschine, sobald die Bedingung am Zustandsübergang erfüllt ist.

In den ersten beiden Fällen muß die Prozeßmaschine mit dem Werkzeug des Benutzers kommunizieren, außerdem die Bedingungen am Zustandsübergang überwachen und die zugeordnete Aktion ausführen. Diese Aufgaben übernimmt ein *FSMAgent*, der in die Prozeßmaschine geladen wird. Der *FSMAgent* lädt wiederum Komponenten, die Bedingungen und Aktionen implementieren. Im *genform*-Framework sind Komponenten für häufig benötigte Bedingungen und Aktionen enthalten. Diese können direkt wiederverwendet oder prozeßspezifisch erweitert werden.

Bedingungen überwachen selbständig den Prozeßzustand in der Objektbank. Sie nutzen hierzu den Benachrichtigungsmechanismus des OMS. Nach einer Änderung des Prozeßzustands wird geprüft, ob sich der Zustand der Bedingung ändert. Falls ja, wird diese Änderung dem *FSMAgent* mitgeteilt, der wiederum den Zustandsübergang freigibt oder direkt ausführt.

Eine Komponente, die eine Aktion implementiert, kann beliebige Operationen auf der Objektbank ausführen. Außerdem kann sie Werkzeugkommandos ausführen, etwa zum Prüfen oder Transformieren eines Dokuments.

Statt des *FSMAgent*, der das Zustandsdiagramm interpretiert, kann auch ein prozeßspezifischer Agent implementiert werden. Im Prozeßmodell wird dann spezifiziert, welcher Agent zur Verwaltung von Dokumentzuständen verwendet werden soll, und die Prozeßmaschine verwendet diesen. Somit kann der Zustandsautomat auch direkt im Prozeßagenten implementiert und auf die Spezifikation per Zustandsdiagramm verzichtet werden. Alternativ können Zustände auch auf andere Art beschrieben werden, etwa mit Petri-Netzen oder *State Charts* [154]. Die Beschreibung wird dann von einem speziellen Agenten interpretiert.

Die Prozeßmaschine wird also auf zwei Ebenen realisiert: Auf der Spezifikationsebene wird das Prozeßmodell graphisch und textuell beschrieben und diese Spezifikation von den generischen Komponenten interpretiert. Auf der Implementierungsebene können Komponenten erweitert oder angepaßt und in die Prozeßmaschine integriert werden.

1.6.6 Prozeßunterstützung in *PI-SET*

PI-SET enthält folgende Prozeßwerkzeuge:

- Mit dem *Planungswerkzeug* können Projekte basierend auf einem gegebenen Prozeßmodell geplant werden. Der Planer nutzt dazu einen graphischen Editor, mit dem er den Projektplan bearbeiten kann.
- Mit dem *Managementwerkzeug* wird der laufende Prozeß als Aufgabennetz dargestellt. Es enthält die Aufgaben mit ihren Zuständen und die Beziehungen zwischen den Aufgaben. Der Manager kann das Aufgabennetz manipulieren, also Aufgaben und Beziehungen erzeugen oder die Eigenschaften der Aufgaben bearbeiten.
- Die Entwickler nutzen eine *Agenda* zur Anzeige ihrer Aufgaben. Über die Agenda kann auch mit der Bearbeitung einer Aufgabe begonnen werden: Der Entwickler kann hier direkt CASE-Werkzeuge auf den zugeordneten Dokumenten starten.

Die Prozeßwerkzeuge stellen unterschiedliche Werkzeugsichten auf den Prozeßzustand dar. Gemeinsam ist ihnen das grundlegende Datenmodell, mit dem Aufgaben und ihre Beziehungen verwaltet werden. Im Planungswerkzeug wird dieses Datenmodell um Informationen wie Aufwände und Ressourcenbedarf erweitert; im Managementwerkzeug ist der aktuelle Aufgabenzustand relevant. Die Agenda enthält nur die Aufgaben eines Benutzers. Neben den Datenmodellen unterscheiden sich auch die Darstellungen und die verfügbaren Kommandos in den verschiedenen Werkzeugsichten: Im Planungswerkzeug wird der Prozeß graphisch dargestellt und es kann eine Netzplanberechnung durchgeführt werden. Die Agenda nutzt hingegen eine baumartige Darstellung und bietet Kommandos zum Starten der zugeordneten Werkzeuge an.

1.6.7 Beitrag der Arbeit

Repository

In der Arbeit wird ein Datenmodell entworfen, mit dem Prozeßinstanzen in einem objektorientierten Datenbank-Managementsystem verwaltet werden können, und das auch Beziehungen zum feingranularen Datenmodell der Dokumente enthält. Das OMS dient als Speicher für den Prozeßzustand, seine Dienste werden von der Prozeßmaschine und den Prozeßwerkzeugen zur Manipulation des Prozeßzustands genutzt. Die Logik zur Manipulation des Prozesses ist in den Komponenten des *genform*-Frameworks implementiert, somit sind die Verwaltung des Prozeßzustands und die Mechanismen zu dessen Manipulation getrennt (ähnlich wie in *Process Wall* [161]). Diese Trennung wäre bei Verwendung eines verhaltensmäßig objektorientierten OMS (wie in [105] gefordert) nicht möglich, da die verwendeten Klassen sowohl das Datenmodell beschreiben als auch die Funktionen zur Manipulation implementieren würden. Verwendet werden der Sichtenmechanismus des OMS, gruppenorientierte Zugriffskontrollen und Transaktionen mit feingranularen Sperren.

Gleichzeitig wird experimentell gezeigt, daß das verwendete OMS in Verbindung mit der vorgestellten Werkzeugarchitektur auch zum Bau prozeßorientierter Umgebungen einsetzbar ist und zahlreiche Dienste bereitstellt, die andernfalls von der PSEU implementiert werden müßten, etwa verteilte Benachrichtigungen und Zugriffsrechte. Problematisch ist die Skalierbarkeit, da von einem zentralen Repository ausgegangen wird, das bei großen Prozessen mit vielen parallel ausgeführten Werkzeugen zum Engpaß werden kann. Allerdings können diese Probleme durch Optimierungen in *genform* und H-PCTE gemildert werden.

Werkzeuge

In der Arbeit wird geprüft, ob der vorgeschlagene Werkzeug-Konstruktionsansatz auch für Prozeßwerkzeuge tauglich ist. Es zeigt sich, daß die Vorteile der Werkzeugkonstruktion mit *genform* auch für Prozeßwerkzeuge genutzt werden können und der Ansatz hinreichend flexibel ist, um auch für diese spezielle Klasse von Werkzeugen nutzbar zu sein. Durch den gemeinsamen Werkzeug-Konstruktionsansatz wird die Integration von CASE- und Prozeßwerkzeugen erleichtert [247].

Prozeßmaschine

In der Arbeit wird vorgeschlagen, wie eine Prozeßmaschine flexibel aus Komponenten zusammengesetzt und über das OMS mit den übrigen Werkzeugen der PSEU integriert werden kann. Wichtig ist hier der Benachrichtigungsmechanismus des OMS, der die Prozeßmaschine über Änderungen des Prozeßzustands informiert. Anhand einer einfachen Prozeßmodellierungssprache wird gezeigt, wie ein konkretes Modellierungsparadigma umgesetzt werden kann. Prozesse werden zunächst graphisch beschrieben; Implementierungsdetails stecken in den Komponenten der Prozeßmaschine.

Fazit

Ziel der Arbeit ist die technische Umsetzung einer Prozeßunterstützung in einer CASE-Werkbank unter Verwendung der vorgestellten Konzepte aus dem Werkzeugbau. Die Prozeßmodellierungssprache dient daher nur zu Demonstrationszwecken. Durch den einfachen

Aufbau der Prozeßmaschine und die flexiblen Erweiterungsmöglichkeiten können PSEU-Prototypen einfach und schnell erstellt werden.

1.7 Gliederung der Arbeit

In Kapitel 2 wird der Hintergrund der Arbeit beleuchtet, dazu gehört der Stand der Technik beim Bau von CASE-Werkzeugen, bekannte PSEU-Ansätze und das verwendete OMS H-PCTE. Kapitel 3 beschreibt den Werkzeugbau mit *genform* und geht detailliert auf Werkzeugschemata und die vorhandenen Werkzeugkomponenten ein. Kapitel 4 erläutert, wie in *genform* Prozesse beschrieben werden und welche Architektur die mit *genform* gebauten PSEU haben. Außerdem wird beschrieben, wie die Objektbank für die Prozeßausführung eingerichtet wird und wie die Integration von Prozeß- und CASE-Werkzeugen funktioniert. Kapitel 5 ist dem Bau von PSEU mit *genform* gewidmet und beschreibt auch die Meta-Werkzeuge dazu. Kapitel 6 faßt die Arbeit zusammen und gibt einen Ausblick auf mögliche Anschlußarbeiten.

Kapitel 2

Hintergrund

In diesem Kapitel wird zunächst der Forschungskontext erläutert, in den sich die Arbeit einordnet. Hierzu zählen *CASE-Werkzeuge* und *CASE-Umgebungen* (Abschnitt 2.1). Es wird ein kurzer Überblick über die CASE-Technologie gegeben und Anforderungen an Datenverwaltungssysteme werden erläutert. Das Vorgehen bei der Auswahl und dem Einsatz von Werkzeugen wird in Abschnitt 2.1.3 beschrieben; die Probleme mit der CASE-Technologie in Abschnitt 2.1.4.

Meta-Umgebungen (Abschnitt 2.2) unterstützen den Bau von CASE-Werkzeugen, die an die Anforderungen unterschiedlicher Organisationen, Projekte und Benutzer angepaßt sind: Sie geben dem Werkzeugentwickler die Möglichkeit, angepaßte Werkzeuge zu bauen – und dies einfacher, effizienter und auf höherer Abstraktionsebene als es mit herkömmlichen Programmiersprachen möglich ist. *Meta-CASE-Systeme* (Abschnitt 2.2.2) arbeiten mit einer Beschreibung der verwendeten Software-Entwicklungsmethode. Sie beschränken sich aber meist auf die Unterstützung der darin definierten Diagrammtypen.

Prozeßorientierte Umgebungen sind eine spezielle Klasse von Meta-Umgebungen. Sie werden mit einer Beschreibung des Software-Entwicklungsprozesses parametrisiert und bieten eine Unterstützung bei der Modellierung, Analyse und Ausführung von Prozessen. Abschnitt 2.3 geht auf einige technische Eigenschaften von PSEU ein und beschreibt die Anforderungen von PSEU an Datenverwaltungssysteme. Danach werden in Abschnitt 2.3.4 einige Datenverwaltungssysteme für PSEU kurz vorgestellt.

Durch die Kombination von Meta-CASE-Systemen und prozeßorientierten Umgebungen können Werkzeuge sowohl an die benötigten Konzepte und Notationen, als auch an das gewünschte Vorgehen angepaßt werden – hierauf geht Abschnitt 2.4 näher ein. In Abschnitt 2.5 wird schließlich das verwendete Objektverwaltungssystem H-PCTE beschrieben und der ISO-Standard PCTE, auf dem es basiert.

2.1 CASE-Werkzeuge und Umgebungen

Mit der zunehmenden Verbreitung immer komplexerer Software-Systeme in allen Bereichen des täglichen Lebens wächst auch die Bedeutung der Werkzeuge, mit denen diese Systeme geplant, entworfen, entwickelt, getestet und gewartet werden: *"Software engineering tools*

and environments are therefore becoming increasingly important enablers, as the demands for software, and its complexity, grow beyond anything that was imagined at the inception of this field a few decades ago." [156]

Wichtig für eine effiziente Nutzung der Werkzeuge ist, daß diese untereinander *integriert* sind [319], also auf gemeinsamen Daten arbeiten, ihre Funktionen anderen Werkzeugen zur Verfügung stellen und eine einheitliche Benutzungsschnittstelle bieten. Eine Sammlung integrierter Werkzeuge wird als *Software-Entwicklungsumgebung* (SEU) bezeichnet [202].

Computer-Aided Software Engineering (CASE) wird oft mit den frühen Software-Entwicklungsphasen in Verbindung gebracht [253]. Fugetta definiert CASE allgemeiner als *"computerized applications supporting and partially automating software-production activities"* [121].

Er unterscheidet folgende Kategorien von CASE-Produkten:

- Ein *CASE-Werkzeug* ist ein Programm, das eine bestimmte Aufgabe im Software-Entwicklungsprozeß unterstützt.
- Eine *CASE-Werkbank* (*workbench*) integriert mehrere Werkzeuge in einer Anwendung. Sie unterstützt damit bestimmte Tätigkeiten im Software-Entwicklungsprozeß.
- Eine *CASE-Umgebung* ist eine Sammlung von Werkzeugen und Werkbänken, die den gesamten Software-Entwicklungsprozeß unterstützt.

Eine CASE-Umgebung muß daher mehr umfassen als nur die eigentlichen Produktionswerkzeuge [42, 81, 121, 244, 314, 315]. Henderson und Cooprider [163] unterscheiden die drei Funktionsbereiche:

1. *Production Technology* mit den Unterbereichen *Representation* (Darstellung und Manipulation von Dokumenten), *Analysis* (Konsistenz- und Qualitätsprüfungen) und *Transformation* (Erzeugen von abgeleiteten Dokumenten).
2. *Coordination Technology* mit den Unterbereichen *Control* (Zuweisen von Aufgaben zu Bearbeitern und Überwachung der Ausführung, Einschränkung der möglichen Aktionen von Benutzern etwa durch Zugriffsrechte) und *Cooperative Functionality* (Austausch von Informationen).
3. *Organizational Technology* unterstützt die beiden anderen Funktionsbereiche mit den Unterbereichen *Infrastructure* (Vorgehensweisen und Qualitätsstandards) und *Support* (organisatorische Richtlinien, Hilfesysteme, Unterstützung bei der Einarbeitung).

Neben der Bearbeitung von Software-Produkten muß eine CASE-Umgebung also auch Funktionen anbieten, mit denen Benutzer erforderliche Aufgaben identifizieren und planen können, und die sie bei der korrekten und effizienten Ausführung dieser Aufgaben unterstützen [314]. Weitere wichtige Aspekte sind die Kooperation [73] und Kommunikation [315] mehrerer Entwickler.

2.1.1 Werkzeugintegration

Wesentliche Eigenschaft einer SEU ist, daß die enthaltenen Werkzeuge untereinander *integriert* sind, also in der SEU einen größeren Nutzen erbringen als die Summe der einzelnen Werkzeuge. Folgende Arten der Werkzeugintegration werden unterschieden [305]:

Benutzungsschnittstellen-Integration

Durch das einheitliche Aussehen und Verhalten der (graphischen) Benutzungsschnittstellen (GUI) kann der Benutzer Kenntnisse in der Bedienung eines Werkzeugs auf andere Werkzeuge übertragen. Auch die Paradigmen, die der Interaktion mit den Werkzeugen zugrundeliegen, sollten in den verschiedenen Werkzeugen gleich sein (z.B. baumartige Komponentenstrukturen versus Hypertext-Strukturen).

Steuerungsintegration

Durch Steuerungsintegration können die von einzelnen Werkzeugen angebotenen Dienste und Kommandos flexibel innerhalb der SEU kombiniert werden. Eine grobgranulare Steuerungsintegration kann mit *envelopes* erreicht werden. Ein *envelope* kapselt ein Werkzeug und dient als Adapter zwischen den Schnittstellen des Werkzeugs und der SEU [88]. Bei einer nachrichtenbasierten Integration [280] können einzelne Dienste eines Werkzeugs von der SEU aufgerufen werden; umgekehrt informiert das Werkzeug diese über den Aufruf von Operationen und deren Ergebnisse. Vorausgesetzt, alle Werkzeuge verwenden ein gemeinsames Protokoll und implementieren alle nötigen Nachrichten.

Datenintegration

Werkzeuge arbeiten auf einem gemeinsamen, konsistenten Datenbestand, wobei sie verschiedene *Sichten* auf die Daten verwenden können [242]. Wichtige Kriterien für die Datenintegration von Werkzeugen sind Interoperabilität, die Vermeidung von Redundanz, die Erhaltung der Konsistenz der Daten, die Möglichkeit des Datenaustauschs zwischen Werkzeugen und die Synchronisation der Daten.

Eine zentrale Datenbank, in der alle Daten einer Umgebung verwaltet werden, erleichtert die Datenintegration [282]: Die Daten werden zentral verwaltet, Werkzeuge können sowohl eigene Daten bearbeiten, als auch auf die Daten anderer Werkzeuge zugreifen und somit über die Datenbank kommunizieren [319]. Auch können Konsistenzbedingungen leichter geprüft werden. Der PCTE-Standard [174] definiert eine Schnittstelle, über die alle Werkzeuge einer Umgebung auf die zentrale Datenbank zugreifen können.

Arbeiten die Werkzeuge nicht auf einer zentralen Datenbank, wird die Datenintegration schwieriger: In jedem Werkzeug ist eine Komponente zur Datenverwaltung enthalten, die meist eigene Speicherformate verwendet. Um Daten zwischen Werkzeugen auszutauschen, müssen also Konverter implementiert werden [41]; die Konsistenz zwischen Daten verschiedener Werkzeuge kann so nur sehr schwierig gesichert werden.

Austauschformate reduzieren die Schwierigkeiten, da nicht alle Datenformate untereinander konvertiert werden müssen, sondern nur jedes Werkzeug eine Im- und Exportfunktion für das Austauschformat anbieten muß. Hess und Schulz beschreiben in [169] ein proprietäres Austauschformat auf Basis von XML. Die Daten werden mit Parsern aus den verschiedenen Dokumenten des Software-Lebenszyklus erzeugt und in einem Hypertext-Repository gespeichert. *XMI* (*XML Metadata Interchange*) [83] ist ein standardisiertes Austauschformat, das auch Informationen über die Semantik der Daten enthält, und damit die Weiterverarbeitung durch den Empfänger erleichtert. In [307] werden die Erfahrungen bei der Verwendung von XMI und dem älteren *CASE Data Interchange Format* (*CDIF*) [49] verglichen. Die Austauschformate werden hier mit verschiedenen Reengineering-Werkzeugen genutzt.

2.1.2 Datenverwaltung

SEU stellen spezielle Anforderungen an Datenverwaltungssysteme [33, 71], die über die Anforderungen etwa von kaufmännischen Anwendungen hinausgehen. Die Verwendung eines *Datenbank-Managementsystems* (DBMS) zur Datenverwaltung hat folgende Vorteile [22]:

- *Datenintegration*: DBMS bieten eine gemeinsame Schnittstelle für den Zugriff auf die Daten und ihre Manipulation an. Diese Schnittstelle wird von allen Werkzeugen der SEU genutzt.
- *Orientierung an der Anwendung*: In den Datenbankschemata kann die Semantik der Daten zum großen Teil ausgedrückt werden.
- *Datenintegrität*: Der konsistente Zustand der Daten wird garantiert und auch nach einem Fehler wieder hergestellt.
- *Komfortabler Zugriff*: DBMS bieten mächtige Schnittstellen für den Zugriff auf die gespeicherten Daten.
- *Datenunabhängigkeit*: Die interne Struktur der Daten wird vor den Anwendungen geheimgehalten, so daß Änderungen der Datenstrukturen nur minimale Auswirkungen auf die Anwendungen haben.

Die zentrale Datenbank, die alle Daten einer Entwurfsumgebung enthält, wird auch *Repository* [34] genannt. Ein Repository kann zwar auf Basis eines Dateisystems oder eines relationalen DBMS implementiert werden – allerdings liegt es nahe, ein möglichst mächtiges Datenverwaltungssystem zu nutzen und fehlende Dienste oberhalb in einem *Repository Manager* [34] zu implementieren.

Objektorientierte Datenbank-Managementsysteme (OODBMS) [310] sind gut für die Verwaltung der komplex strukturierten Daten einer SEU geeignet [102, 105]. Sie unterstützen die (parallele) Arbeit mehrerer Werkzeuge und Benutzer durch Sichten, Zugriffsrechte, spezielle Transaktionskonzepte und Mechanismen zur Konsistenzsicherung. OODBMS werden im folgenden auch als *Objektverwaltungssysteme* (*Object Management System*, OMS) bezeichnet.

Verwaltung feingranularer Dokumente

Softwaredokumente haben meist eine komplexe Struktur. Diese sollte feingranular modelliert und im OMS in Form eines abstrakten Syntaxgraphen repräsentiert werden können [105]. Hierdurch können Konsistenzbedingungen innerhalb eines Dokuments und zwischen mehreren Dokumenten überwacht werden. Außerdem wird der Zugriff auf einzelne Dokumentteile effizienter, da nur die betroffenen Teile gelesen und geschrieben werden müssen [294]. Kommerzielle OODBMS wie O_2 [51] (z.B. in [100]) und *Gemstone* [45] (z.B. in [103]) und Prototypen wie *GRAS* [211] (z.B. in [74]) oder H-PCTE [201] (z.B. in *ToolFrame* [69]) unterstützen die feingranulare Datenmodellierung.

Konsistenzsicherung

Ein Datenbestand befindet sich in einem konsistenten Zustand, wenn bestimmte, vorher festgelegte Bedingungen erfüllt sind. Welche Bedingungen dies sind, wird durch den Werkzeugentwickler festgelegt. Das OMS sollte die Definition von Konsistenzbedingungen ermöglichen

und diese bei der Manipulation des Datenbestands prüfen. Neben Konsistenzbedingungen, die stets erfüllt sein müssen (z.B. Eindeutigkeit von Schlüsseln) kann auch die temporäre Verletzung von Konsistenzbedingungen zulässig sein, da andernfalls die Bearbeitung eines Dokuments nicht möglich wäre (z.B. Kardinalität einer Beziehung). Das OMS sollte Mechanismen anbieten, mit denen Werkzeuge (automatisch infolge einer Benachrichtigung oder auf explizite Anforderung) über eventuelle Konsistenzverletzungen informiert werden können [294].

Die Konsistenzsicherung kann auch an externe Werkzeuge (*Analysatoren*) delegiert werden, die beliebige Konsistenzbedingungen prüfen. Sie können dabei auf Dienste des OMS wie Abfragesprachen zurückgreifen [166].

Zugriffsrechte

Da an einer Software-Entwicklung meist mehrere Entwickler beteiligt sind, sollte das OMS (möglichst hierarchische) Benutzergruppen verwalten, denen Zugriffsrechte an einzelnen Objekten im OMS gewährt werden können. In Verbindung mit einer feingranularen Modellierung der Dokumente können so auch Zugriffsrechte an einzelnen Dokumenteinträgen vergeben werden. PCTE [317] bietet hierarchische Benutzergruppen und Zugriffsrechte auf Basis von *Access Control Lists*, die Objekten zugeordnet werden (Abschnitt 2.5.3).

Sichten

Eine *Sicht* (oder ein externes Schema) enthält einen Ausschnitt aus dem konzeptuellen Datenbankschema. Greift ein Werkzeug über eine Sicht auf die Datenbank zu, kann es also nur eine Teilmenge der Daten manipulieren. Somit können nicht relevante Daten (oder Details der Daten) ausgeblendet und vor unerlaubtem Zugriff geschützt werden.

Technisch kann durch Sichten auch eine redundante Speicherung von Informationen vermieden werden: Statt die für den Entwurf und die Implementierung relevanten Informationen einer Klasse in verschiedenen Objekten zu speichern, wird die Klasse durch ein Objekt im Repository repräsentiert, auf das mit einer Entwurfs- und einer Implementierungssicht zugegriffen werden kann.

Konzeptionell werden bei der Modellierung verschiedene Aspekte eines Systems, wie Daten, Zustände und Verhalten, in verschiedenen Sichten beschrieben. Wichtig ist hier die Konsistenz der verschiedenen Sichten [120], für deren Gewährleistung wiederum die technischen Möglichkeiten eines OMS genutzt werden können.

Lange Transaktionen

Die 'Arbeitspakete', die ein Software-Entwickler durchführt, sind meist komplex strukturiert, überdauern einen Zeitraum von Stunden bis Wochen, erfordern den Austausch von Zwischenergebnissen und die Interaktion mit dem Benutzer [218]. Folglich sind herkömmliche Transaktionskonzepte für SEU nicht geeignet [22, 33]: Das OMS sollte einen flexiblen, adaptierbaren Transaktionsmechanismus anbieten, der mit der Konfigurations- und Versionsverwaltung integriert ist [23].

Lange Transaktionen können auch als *Sub-Datenbanken* (oder Arbeitsbereiche, *workspaces*) betrachtet werden, also private und isolierte Arbeitsbereiche, die die Daten einer Transaktion

enthalten. Sub-Datenbanken können auch physisch verteilt sein. Die Konfigurationsverwaltungssysteme *ADELE* [108] und *EPOS* [252] bieten Arbeitsbereiche. In *Raleigh* [196] werden Aufgaben mit Transaktionen gleichgesetzt. *Coordinated Activities* setzen Sperren auf den bearbeiteten Objekten im gemeinsamen OMS, so daß die Isolation gewährleistet ist.

Trigger

Trigger sind Prozeduren, die vom OMS als Reaktion auf Ereignisse wie das Ändern, Erzeugen oder Löschen von Objekten ausgeführt werden. Sie können genutzt werden,

- um Folgeoperationen auszuführen, etwa das Setzen von Attributwerten nach dem Erzeugen eines Eintrags.
- um verschiedene Dokumente zu integrieren: Änderungen an einem Dokument werden auf andere Dokumente übertragen. Auf diese Weise kann beim Erzeugen eines Eintrags im ER-Diagramm ein passender Eintrag im zugehörigen Datenlexikon erzeugt werden.
- um Konsistenzbedingungen zu garantieren, indem auf die Verletzung einer Konsistenzbedingung mit geeigneten Gegenmaßnahmen reagiert wird.

Versions- und Konfigurationsverwaltung

Während der Software-Entwicklung werden Dokumente erzeugt und bearbeitet. Dabei entstehen zum einen verschiedene *Varianten* eines Dokuments, also alternative Ausprägungen, die sich in bestimmten Eigenschaften unterscheiden und (für eine bestimmte Zeit) parallel existieren [84]. Verschiedene *Revisionen* ergeben sich, wenn die aktuelle Version eines Dokuments durch eine weiterentwickelte Nachfolge-Version ersetzt wird [65]. Neben diesen Versionen einzelner (zum Teil komplexer) Objekte müssen oft auch Mengen oder *Konfigurationen* von Objekten betrachtet werden, die in Beziehung zueinander stehen – etwa alle Objektdateien einer Programmversion. Dies ist Aufgabe der Konfigurationsverwaltung.

Das OMS muß Operationen zur Versions- und Konfigurationsverwaltung anbieten. Die Mechanismen sollten dabei hinreichend flexibel sein, so daß der Werkzeugentwickler die für sein Werkzeug am besten geeigneten Verfahren einsetzen kann [294]. Dies wurde z.B. in PCTE [317] berücksichtigt.

Undo-/Redo-Mechanismus

Durch Undo- und Redo-Mechanismen werden Änderungen im OMS zurückgesetzt bzw. wiederholt. Der Undo-Mechanismus wird genutzt, um nach dem Abbruch einer Transaktion bereits durchgeführte Änderungen rückgängig zu machen. In H-PCTE wird eine Schnittstelle angeboten, über die Werkzeuge einzelne Änderungen explizit zurücksetzen können [270]. Im Werkzeug selbst müssen die durchgeführten Kommandos nicht protokolliert werden, da das OMS einen *Änderungs-Log* führt. Das Werkzeug muß lediglich *Sicherungspunkte* setzen, um Sequenzen von Änderungsoperationen zu kennzeichnen, die durch den Aufruf der Undo-Operation rückgängig gemacht werden sollen.

Abfragesprache

Mit (deskriptiven) *Abfragesprachen* können Informationen über die Daten im OMS mittels (*ad hoc*-) Abfragen ermittelt werden [310]. Sie stellen damit einen großen Vorteil gegenüber

Netzwerk- und hierarchischen DBMS dar, auf die nur *navigierend* zugegriffen werden kann.

Nach Henrich et al. [167] werden in SEU drei Arten von Zugriffsverfahren benötigt:

- Navigierender Zugriff über Beziehungen zwischen Objekten
- Mengenorientierter Zugriff durch SQL-artige Abfragen
- Dokumenten-Retrieval zum Finden von Informationen in Freitexten

Abfragen können verwendet werden, um bestimmte Objekte in der Gesamtmenge zu finden, und mit diesen weiterzuarbeiten (z.B. alle Klassen mit leerem Spezifikationsfeld). Abfragen können auch Informationen aus Dokumenten herausfiltern, etwa die Anzahl der Klassen eines Projekts. Abfrageergebnisse können auch bei der Transformation genutzt werden, etwa beim Generieren von Reports. Schließlich sind Abfragen nützlich, um Konsistenzbedingungen zu prüfen und Fehler in Dokumenten zu finden: *ToolFrame* [69] verwendet die Abfragesprachen *P-OQL* [164] und *Ntt* [152] in Suchfunktionen [245] und zum Prüfen von Konsistenzbedingungen [166].

Verteilung

Software-Projekte werden meist von mehreren Personen durchgeführt. Die Arbeitsplätze der Beteiligten können sich dabei an unterschiedlichen Orten befinden. Um allen Beteiligten den Zugriff auf die gemeinsamen Projektdaten zu ermöglichen, sind zwei Wege denkbar [23]: Die Daten können in einer zentralen Datenbank verwaltet werden, auf die alle Mitarbeiter von ihren verteilten Rechnern aus zugreifen (*Client-Server*-Ansatz). Dies kann bei großen Projekten schnell zu einer Überlastung des zentralen OMS führen. Alternativ können die Daten auf mehrere OMS verteilt werden, auf die Benutzer lokal zugreifen. Die Verteilung sollte für den Benutzer transparent sein, so daß er nicht wissen muß, an welchem Ort die Daten gelagert werden. In PCTE [317] werden Daten auf *volumes* verwaltet, die auf verschiedenen Rechnern innerhalb eines Netzwerks liegen können.

Schema-Evolution

Da ein Software-Projekt meist über einen längeren Zeitraum hinweg durchgeführt wird, können sich innerhalb dieser Zeit auch Anforderungen und Randbedingungen ändern, was wiederum eine Anpassung der verwendeten Datenbankschemata erforderlich machen kann – etwa die Erweiterung von Objekttypen um neue Attribute und Beziehungen. Die bereits vorhandenen Instanzen der geänderten Typen müssen in diesem Fall aktualisiert werden; entweder sofort oder erst beim nächsten Zugriff (*immediate* vs. *lazy update*) [294]. Bei komplexen Typänderungen kann es dabei nötig sein, geeignete Transformationsfunktionen zur Konvertierung der vorhandenen Objekte zu implementieren (wie in *TESS* [224]).

2.1.3 Auswahl und Einsatz

Aufgrund der Tatsache, daß Werkzeuge stets sehr speziell auf ein bestimmtes Einsatzgebiet ausgerichtet sind (*"each tool or environment is still highly specific to some context"* [156]), kommt der Auswahl der Werkzeugunterstützung eine große Bedeutung zu. Sie ist nach Thayer [304] Teil der Planungsphase eines Projekts.

Wird der Bedarf nach einem neuen Werkzeug festgestellt, müssen zunächst Anforderungen definiert und Auswahlkriterien festgelegt werden [277]. Mögliche Kandidaten können mit Hilfe von Vorführungen, Checklisten oder Piloteinsätzen bewertet werden. Neben den Werkzeugeigenschaften spielen auch andere Faktoren wie die Unterstützung durch den Hersteller, Finanzen und die Auswirkungen des Werkzeugeinsatzes auf die Abläufe in er eigenen Organisation eine Rolle.

Bruckhaus [43] definiert mit *TIM* (*Tool Insertion Method*) eine Methode, mit der Werkzeuge in Organisationen eingeführt werden können – mit dem Ziel, den Software-Entwicklungsprozeß zu verbessern. Auf die Auswahlphase folgen in TIM die Schritte:

1. *Customize*, also die Anpassung des Werkzeugs, so daß die angestrebten Prozeßverbesserungen auch erreicht werden können.
2. *Validate*, also die Bewertung des Werkzeugeinsatzes und die iterative Verbesserung von Werkzeug und Prozeß.
3. *Implement*, also der Einsatz von Werkzeug und Prozeß in der täglichen Arbeit. Hierzu zählt auch die Schulung der Mitarbeiter und die Bereitstellung von technischem Personal zur Unterstützung der Benutzer.

Festzuhalten bleibt, daß die Auswahl geeigneter Werkzeuge entscheidend für den Erfolg eines Projekts oder einer ganzen Organisation sein kann, daß der Auswahlprozeß aufwendig und teuer ist und nur funktioniert, wenn die Anforderungen an die Werkzeuge nicht allzu weit von den Vorstellungen der Werkzeughersteller entfernt sind, die Werkzeuge also die Anforderungen der Benutzer bereits erfüllen oder angepaßt werden können. Aus diesem Grund ist in der Auswahlphase eine Alternative vorgesehen, die zur Eigenentwicklung des benötigten Werkzeugs führt.

2.1.4 Probleme mit CASE

Die Werkzeuge für die späten Software-Entwicklungsphasen sind weitgehend ausgereift und es herrscht Übereinstimmung darüber, wie Compiler, Testwerkzeuge und Editoren (von persönlichen Präferenzen abgesehen) aussehen und funktionieren sollen.

Die in den frühen Phasen verwendeten Methoden hängen von der Art des Projektes ab. Beim Entwurf von Informationssystemen wird z.B. die objektorientierte Modellierung eingesetzt; für Echtzeitsysteme ist die Zustandsmodellierung mit Petri-Netzen und Zustandsdiagrammen besser geeignet. Oft werden auch an die speziellen Projekterfordernisse angepaßte Varianten von Methoden verwendet. Die Zahl solcher speziellen Methoden wird in Folge der wachsenden Zahl neuer Anwendungsgebiete wie *E-Commerce* und *Pervasive Computing* in Zukunft weiter wachsen [70] – ebenso der Bedarf nach passenden Werkzeugen.

Dies legt den Schluß nahe, daß die Probleme mit CASE-Werkzeugen, die Studien in den letzten Jahren ermittelt haben, auch in naher Zukunft nicht gelöst sein werden:

– Die hohe Komplexität verstellt den Blick auf den tatsächlichen Nutzen der CASE-Werkzeuge [173].
– Der hohe Einarbeitungsaufwand schreckt die Benutzer ab und verursacht hohe Kosten.

– Nach der Einführung der Werkzeuge sinkt oftmals die Produktivität, statt wie erhofft zu steigen [210].
– Entwickler sind nicht sehr stark motiviert, verfügbare Werkzeuge auch tatsächlich zu verwenden, sofern dies nicht vom Management vorgeschrieben wird.

Enttäuschte Erwartungen und mangelnde Motivation führen dazu, daß viele Werkzeuge, die den Entwicklern zur Verfügung stehen, nicht oder nur für kurze Zeit eingesetzt werden. Wenn die Werkzeuge eingesetzt werden, so bleiben viele der Funktionen ungenutzt [222], etwa solche zur Analyse oder Transformation von Dokumenten. Es wird also nicht nur Zeit und Geld verschenkt, sondern es bleiben auch Chance zur Qualitätsverbesserung und zur Steigerung der Produktivität ungenutzt.

Gegenmaßnahmen

Es können nur solche Werkzeuge sinnvoll eingesetzt werden, die an die konkreten Einsatzbedingungen angepaßt sind. Da sich die Einsatzbedingungen stark unterscheiden, müssen also die Werkzeuge flexibel an die jeweilige Situation angepaßt werden können [180].

Dies umfaßt nicht nur die Anpassung an die gewählte Methode, sondern auch an die Anforderungen bestimmter Aufgaben und Rollen im Software-Entwicklungsprozeß. Derart angepaßte Werkzeuge sind auch weniger komplex und somit einfacher und schneller einsetzbar, was die Akzeptanzschwelle beim potentiellen Anwender reduziert.

Der Nutzen von Werkzeugen und damit die Motivation der Benutzer kann auch erhöht werden, wenn Teilaufgaben automatisch ablaufen, somit der Benutzer von Routinetätigkeiten entlastet wird, etwa bei der Versionsverwaltung oder der Qualitätssicherung. Umgekehrt sollte die Verwendung bestimmter Funktionen im Software-Entwicklungsprozeß vorgeschrieben werden können, etwa zur Konsistenzprüfung von Dokumenten.

Viele CASE-Werkzeuge enthalten ein vom Werkzeughersteller festgelegtes Vorgehensmodell, das die Werkzeugbenutzer bei der Durchführung ihrer Aufgaben anleitet. Die Werkzeuge werden daher auch als *methodology companions* [314] bezeichnet und können nach verschiedenen Verfahren arbeiten:

– Beim *restriktiven* Ansatz wird das Vorgehen durch das Werkzeug vorgeschrieben, Abweichungen sind nicht möglich. So werden Prüfungen automatisch durchgeführt, gefundene Fehler müssen zuerst behoben werden, bevor im Prozeß fortgefahren werden kann. In [58] sind 26 solcher Prüfungen (*model checks*) für OOA angegeben.
– Beim *führenden* Ansatz werden dem Benutzer Vorschläge unterbreitet, wie eine Aufgabe ausgeführt werden sollte; Prüfungen werden vom Benutzer initiiert und die Ergebnisse als Warnungen verstanden.
– *Flexible* Werkzeuge schließlich bieten dem Benutzer völlige Freiheit bei der Bearbeitung seiner Aufgaben, er kann nach seinem eigenen 'Prozeß' vorgehen und auf Wunsch nachträglich prüfen, ob seine Ergebnisse korrekt und konsistent sind.

Die Eignung der verschiedenen Ansätze hängt zum einen von den Kenntnissen und der Erfahrung der jeweiligen Benutzer ab und zum anderen von der durchzuführenden Aufgabe: Je anspruchsvoller die Aufgabe und je größer die Kenntnisse und Erfahrungen der Benutzer, desto weniger Vorschriften und Restriktionen sollten vorhanden sein. Als Anforderung an

die Werkzeuge ergibt sich hieraus, daß auch die Prozeßunterstützung aufgaben- und rollenspezifisch adaptierbar sein muß. Bei den in [314] untersuchten Werkzeugen war der gewählte Ansatz zur Methodenunterstützung zudem nicht in allen Bereichen konsistent umgesetzt. Gerade die konsistente Umsetzung eines Vorgehensmodells im Werkzeug ist aber erstrebenswert, da Methodenhandbücher nur schlecht zur Anleitung der Entwickler einsetzbar sind [170].

Eine angepaßte Werkzeugunterstützung bietet also die benötigten Dokumenttypen, Darstellungen und Kommandos an und berücksichtigt auch das gewählte Vorgehensmodell. Da sich die genannten Parameter in verschiedenen Organisationen, Projekten und Aufgaben stark unterscheiden, gestaltet sich die Suche nach einer solchen Werkzeugunterstützung schwierig und führt entweder zu einem Kompromiß oder zum Bau maßgeschneiderter Werkzeuge – mehr dazu im nächsten Abschnitt.

2.2 Meta-Umgebungen

Passende CASE-Werkzeuge und Umgebungen sind also einerseits sehr wichtig für die effiziente Produktion von Software mit hoher Qualität. Andererseits sind CASE-Werkzeuge selbst komplexe Software-Produkte und folglich aufwendig und teuer herzustellen.

Allerdings muß eine 'passende' Werkzeugunterstützung für ein Projekt in einem anderen nicht ebenso gut einsetzbar sein: Organisationen, Projekte, zu entwickelnde Systeme und auszuführende Aufgaben unterscheiden sich zu stark. Somit unterscheiden sich auch die Anforderungen an die Werkzeuge – und sind zum Teil sogar noch unbekannt. Einzige Lösung ist die evolutionäre Entwicklung, Verwendung und Verbesserung von Werkzeugen. Diesem Ansatz steht allerdings der hohe Kostenaufwand für die Werkzeugentwicklung gegenüber.

Einen Ausweg weisen *Meta-Umgebungen*: Sie ermöglichen den Bau von CASE-Werkzeugen unter Verwendung mächtigerer Konstrukte, als sie Programmiersprachen bieten. Meta-Umgebungen werden auch als *generische Umgebungen* oder *Umgebungsgeneratoren* bezeichnet; ersteres zielt eher auf die mögliche Anpassung der Werkzeuge an unterschiedliche Anforderungen, während letzteres den Produktionsprozeß der Werkzeuge betont.

Nach Karrer und Scacchi [195] bestehen Meta-Umgebungen aus einem *Konstruktionsmodell* für Werkzeuge, einer Möglichkeit, eine Instanz des Konstruktionsmodells (die *Umgebungsspezifikation*) in eine konkrete Umgebung zu *transformieren*, und einem *Konstruktionsprozeß*, der die nötigen Schritte zum Bau einer Umgebung definiert.

Es gibt verschiedene Ansätze für Meta-Umgebungen:

- *Frameworks* definieren eine Schnittstelle zu grundlegenden Diensten wie Datenverwaltung, Verwaltung von (Betriebssystem-) Prozessen und zum Teil auch für Benutzungsschnittstellen. PCTE (s. Abschnitt 2.5) ist ein Vertreter dieser Klasse.
- *Adaptierbare Umgebungen* (*customizable environments*) enthalten komplexe Funktionen, die an die gegebenen Anforderungen angepaßt werden können. Charakteristisch für adaptierbare Umgebungen ist der Kompromiß zwischen einfacher Verwendung in einem beschränkten Anwendungsgebiet und hoher Flexibilität bei höherem Instantiierungsaufwand.

– Mit der *Prozeßmodellierung* und *Prozeßprogrammierung* können Software-Prozesse spezifiziert und in *prozeßorientierten* Umgebungen (PSEU) ausgeführt werden. Mehr zu PSEU in Abschnitt 2.3.
– Mit Ansätzen zur *Werkzeugintegration* können Werkzeuge so kombiniert werden, daß sie zusammen die Funktionen der Umgebung realisieren (s. Abschnitt 2.1.1).

Bei den Nutzern einer Meta-Umgebung können folgende *Rollen* unterschieden werden: Der *Werkzeugentwickler* nutzt die Meta-Technologie für den Bau und die Anpassung von Umgebungen. Er muß hierzu die Einsatzsituation analysieren, die passende Umgebung spezifizieren (unter Verwendung des Konstruktionsmodells der Meta-Umgebung) und aus der Spezifikation die konkrete Umgebung erzeugen. Die Tätigkeit des Werkzeugentwicklers ist also eine anspruchsvolle – sie umfaßt Daten- und Prozeßmodellierung ebenso wie die Werkzeugintegration.

Der *Komponentenentwickler* baut Werkzeugkomponenten und stellt sie den Werkzeugentwicklern zur Verfügung. Die Anforderungen an die Komponenten legt der Komponentenentwickler zum Teil selbst fest; zum Teil kommen sie von den Werkzeugentwicklern. Eine große Bedeutung kommt der Wiederverwendung von Werkzeugkomponenten zu: Der Komponentenentwickler wählt die am besten geeigneten Komponenten aus Bibliotheken oder vorhandenen Umgebungen aus, um sie anzupassen oder zu erweitern. Andererseits stellt er die neuen Komponenten für eine spätere Wiederverwendung zur Verfügung.

Anforderungen an Meta-Umgebungen

Folgende Anforderungen müssen Meta-Umgebungen erfüllen:

1. *Gute Ergebnisse*: Die grundlegendste Anforderung an eine Meta-Umgebung ist, daß sie den Bau guter und angepaßter CASE-Umgebungen unterstützt. Eine *gute* Umgebung erfüllt allgemeine Anforderungen etwa an die Geschwindigkeit, die Integration der Werkzeuge und die Qualität der Benutzungsschnittstelle. Eine *angepaßte* Umgebung muß zusätzlich die Anforderungen erfüllen, die aus den konkreten Einsatzbedingungen resultieren, also die benötigten Funktionen und Darstellungen anbieten.
2. *Einfache Benutzung*: Die Meta-Umgebung sollte von Werkzeugentwicklern mit unterschiedlichem Kenntnisstand verwendbar sein: Einerseits sollten auch Benutzer mit geringen Kenntnissen einfache Werkzeuge realisieren können; andererseits sollten erfahrenen Benutzern mächtige und flexible Mittel zur Verfügung stehen, mit denen Werkzeuge an die jeweiligen Anforderungen angepaßt werden können. Hier sind auch Mechanismen zur Fehlerprüfung und zur Konsistenzsicherung wichtig, die den Werkzeugentwickler bereits früh auf mögliche Fehler in den gebauten Werkzeugen hinweisen.
3. *Evolutionäres Vorgehen*: Da die Anforderungen an CASE-Werkzeuge in vielen Fällen nicht bekannt oder variabel sind, ist ein evolutionäres Vorgehen bei der Werkzeugentwicklung sinnvoll. Die Meta-Umgebung sollte daher die Evolution der Datenmodelle und die inkrementelle Weiterentwicklung der CASE-Umgebung unterstützen.

2.2.1 Software-Entwicklungsmethoden

In den frühen Software-Entwicklungsphasen unterstützen CASE-Werkzeuge die Entwickler beim Erzeugen, Bearbeiten, Prüfen, Analysieren und Transformieren von Software-Spezifikati-

onen im Rahmen der Analyse und des Entwurfs.

Der Erstellung einer solchen Spezifikation liegt eine *Software-Entwicklungsmethode* (kurz: *Methode*) zugrunde, die vorschreibt, welche Modellierungskonzepte verwendet und wie diese notiert und dargestellt werden. Weiterhin schreibt die Methode vor, in welchen Schritten der Entwickler beim Erstellen der Spezifikation vorgehen sollte.

Die Methodendefinition kann also einerseits dem Entwickler als Anleitung dienen, mit welchen Mitteln er das System beschreiben und wie er dabei vorgehen soll. Das Vorgehensmodell kann auch als Grundlage für die Projektplanung genutzt werden, da es elementare Arbeitsschritte und ihre Ergebnisse definiert. Schließlich bestimmt die Methode auch die Anforderungen an die Werkzeugunterstützung. Die Vorgehensmodelle der Lehrbuchmethoden sind allerdings zu unpräzise und zu vage, um den Entwicklern als Anleitung zu dienen. Song und Osterweil beschreiben in [296], wie das Vorgehensmodell der Booch-Methode verfeinert werden muß, um es als Prozeßmodell für eine PSEU nutzen zu können.

Jeusfeld et al. [187] unterscheiden drei Bereiche, in denen Methoden den Entwurf von Informationssystemen unterstützen sollen:

- Methoden definieren Notationen, die zur Beschreibung eines Systems verwendet werden. Beispiele sind die ER-Modellierung [52], die Moderne Strukturierte Analyse [326] oder die UML [265].
- Methoden versuchen, das Wissen eines Anwendungsbereichs zu erfassen und zu strukturieren. Sie definieren Referenzmodelle, die typische Daten, Prozesse und Funktionen zusammen mit den nötigen Konsistenzbedingungen enthalten.
- Methoden haben das Ziel, die unterschiedlichen Anforderungen, die verschiedene Personen oder Rollen an das System stellen [255], zu erfassen und mögliche Konflikte zu lösen: Anforderungsdefinition als gruppendynamischer Prozeß, etwa mit *Joint Application Design* von IBM [15].

Methodendefinition

Eine *Methodendefinition* umfaßt [308]:

- *Konzepte*, die das Wissen repräsentieren, das in der Methode erfaßt, manipuliert und gespeichert wird. In der OOA sind dies Klassen, Attribute, Operationen, Assoziationen, Pakete.
- *Notationen*, mit denen das Wissen für die verschiedenen Teilnehmer im Entwicklungsprozeß aufbereitet wird. OOA-Modelle werden in Klassendiagrammen dargestellt mit vorgegebenen graphischen Symbolen für die verschiedenen Konzepte und Regeln für ihre Anordnung (Subklassen unterhalb der Superklasse). Für manche Konzepte gibt es verschiedene Darstellungen: Pakete können expandiert dargestellt werden, so daß die enthaltenen Klassen sichtbar sind, oder nur als Symbole mit ihrem Namen. Alternativ zur graphischen Darstellung sind auch die Baum-, Listen- und Tabellendarstellung gebräuchlich. Allgemein gibt es zwischen Konzepten und Notationen eine $m : n$-Beziehung [180].

  Zusätzlich können weitere Eigenschaften wie der Bearbeitungszustand einer Klasse oder Hinweise auf Fehler in der Darstellung berücksichtigt werden. Diese Eigenschaften stehen in engem Zusammenhang mit dem zugrundeliegenden Entwicklungsprozeß, da sie

die Zustände von Dokumenten und Einträgen reflektieren, die für den Prozeßfortschritt relevant sind.

– *Vorgehensweisen*, die festlegen, wie das Wissen in den einzelnen Schritten des Entwicklungsprozesses erzeugt und konsumiert wird.

Das *Dokument* (oder Software-Dokument) wird i.f. als Platzhalter für ein beliebiges Software-Produkt verwendet, also auch für eine einzelne Klasse oder Operation. Dies entspricht der Definition in [286]: *"A* document *is a unified collection of information pertaining to a specific subject or related subjects."*

2.2.2 Meta-CASE-Systeme

Meta-CASE-Systeme zählen zu den adaptierbaren Umgebungen. Wo die Flexibilität herkömmlicher CASE-Werkzeuge nicht ausreicht, kommen sie als *customisable CASE tools* [238] zum Einsatz. Meta-CASE-Systeme werden auch als *CASE shells* bezeichnet und enthalten der Definition nach Mechanismen zur Konstruktion von CASE-Werkzeugen für beliebige Methoden [44].

Es existieren einige Meta-CASE-Systeme, zum großen Teil als Forschungsprototypen, einige im kommerziellen Einsatz. Wesentlicher Bestandteil eines Meta-CASE-Systems ist ein *generischer Editor*, mit dem Dokumente verschiedenen Typs bearbeitet werden können [4].

Um eine Methode zu unterstützen, muß zunächst spezifiziert werden, welche Arten von Elementen und Beziehungen in einem Dokument enthalten sind (Konzepte), und wie diese graphisch oder textuell dargestellt werden sollen (Notationen). Der Entwurf von Methoden und die Konstruktion passender Werkzeuge sind Aufgaben des *Method Engineering* [39]. In Abbildung 2.1 sind die Tätigkeiten und Produkte des *Method Engineering* skizziert (vgl. [258]): Im Rahmen des Methodenentwurfs wird zunächst eine Methodendefinition erstellt. Auf Basis dieser Methodendefinition werden Werkzeuge konstruiert, abhängig vom Meta-CASE-System vollautomatisch durch einen Generator oder ergänzt durch manuelle Programmierung. Das Ergebnis sind CASE-Werkzeuge zur Bearbeitung der Software-Dokumente – wobei die verschiedenen Werkzeuge möglichst miteinander integriert sein sollten. Fließen die beim Einsatz der Werkzeuge gesammelten Erkenntnisse zurück in den Methodenentwurf, können Methode und Werkzeuge inkrementell verfeinert werden.

Eine CAME-Umgebung (*Computer Aided Methodology Engineering*-Umgebung) [238] ist eine integrierte Sammlung von Werkzeugen zur Spezifikation von Methoden, zu deren Auswahl und Analyse, sowie zur Aufzeichnung und Auswertung der Erfahrungen beim Einsatz von Methoden.

2.2.3 Modelle

Das von Meta-CASE-Systemen verwendete Meta-Meta-Modell (*M2-Modell*) [187] zur Beschreibung der Konzepte basiert meist auf dem ER-Modell: GOPRR (*graph, object, property, relationship, role*) in *MetaEdit+* [197], EARA (*entity, aggregate, relationship, attribute*) in *Metaview* [113], EER (*extended entity relationship*) in KOGGE [93].

Bei Manson et al. [235] besteht ein universelles M2-Modell aus den Elementen

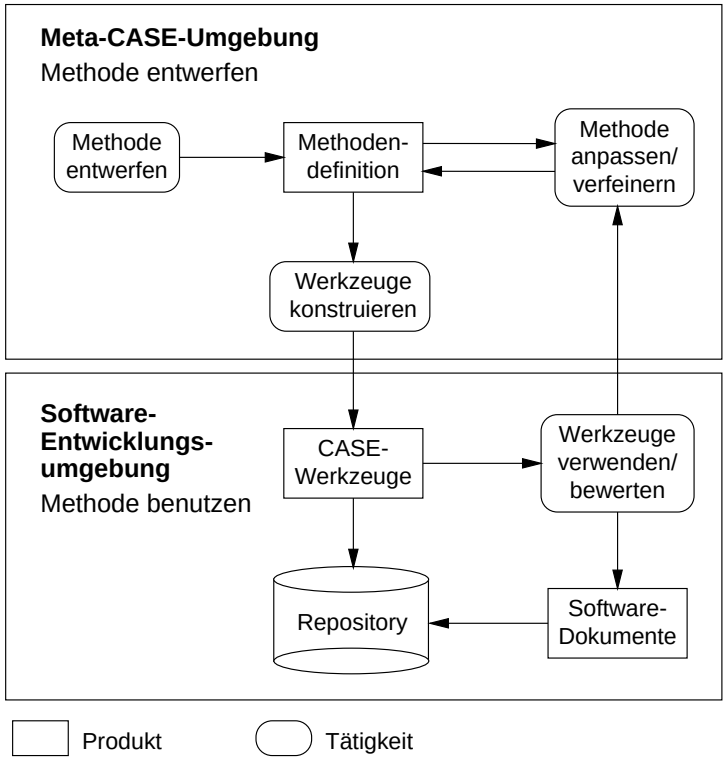


Abbildung 2.1: Einsatz von Meta-CASE-Systemen

- *node*, einer Repräsentation eines eigenständigen Diagramm-Elements,
- *link*, einer Beziehung zwischen genau zwei *nodes* und
- *grouper*, einer Menge von *nodes* und der zugehörigen *links*.

In Abbildung 2.2 sind die Modellierungsstufen mit Beispielen und den eingesetzten Werkzeugen dargestellt (vgl. auch [187]). Die Werkzeuge überspannen jeweils zwei Stufen, angedeutet durch die Klammern in der Abbildung: Die höhere Stufe beschreibt das Modell, mit dem die Werkzeuge arbeiten, die niedrigere die Instanzen des Modells.

Stufe	**Beispiel**	**Werkzeuge**
Meta−Meta−Modell, M2−Modell	*node, link, grouper*	CAME
Meta−Modell, Modellierungssprache	Datenfluß−diagramm (DFD)	Upper CASE
Modell	DFD für Banksystem	Lower CASE
Instanz, Applikation	Banksystem	

Abbildung 2.2: Modellierungsstufen mit Beispielen und Werkzeugen

Den Element- und Beziehungstypen werden Darstellungen (Formulare, Tabellen, graphische Symbole) zugeordnet, durch die sie in Werkzeugen sicht- und manipulierbar werden. Zusätzlich erlauben viele Systeme die Definition von Reports, mit denen Modelle in textuelle

Darstellungen transformiert werden können. Mit speziellen Sprachen werden Konsistenzbedingungen formuliert, die bei der Bearbeitung von Modellen überwacht werden (z.B. [94]) und sich auch auf die graphische Darstellung beziehen können.

Durch die Spezifikation auf hoher Abstraktionsebene können Werkzeuge mit Meta-CASE-Systemen sehr effizient konstruiert werden. Probleme treten auf, wenn die zur Adaptierung eines Werkzeugs nötigen Änderungen im Meta-CASE-System nicht vorgesehen sind [110]. In diesem Fall steigt der Aufwand für die Adaptierung des Werkzeugs überproportional, da die Einschränkungen des Meta-Systems (falls überhaupt möglich) umgangen werden müssen. Kann das Meta-CASE-System durch Routinen erweitert werden, die in der gleichen Programmiersprache wie das Meta-CASE-System selbst geschrieben sind (wie *TBK/Toolbuilder* [5]), müssen die Erweiterungen auf niedrigem Abstraktionsniveau und mit sehr eingeschränkten Möglichkeiten zur Wiederverwendung kodiert werden.

Frameworks zur Konstruktion von CASE-Werkzeugen wie *MetaMOOSE* [110] und *Serendipity* [145] haben dieses Problem nicht, da die objektorientierten Programmiersprachen Wiederverwendung und Abstraktion direkt unterstützen. Nachteile der genannten Ansätze sind die steile Lernkurve, da der Werkzeugentwickler zunächst das Framework verstehen muß, bevor er ein Werkzeug konstruieren kann, und die hohe Komplexität des Frameworks: In *MViews* [147] sind alle Dienste zur Verwaltung von Daten und Modellen, für Verteilung, Zugriffsschutz und Änderungspropagation im Framework selbst implementiert.

Werkzeugvarianten

Nur wenige Ansätze beschäftigen sich mit der Frage, wie leicht *Varianten* eines Werkzeugs erzeugt werden können, die jeweils auf eine bestimmte Benutzungssituation ausgelegt sind und sich nur geringfügig unterscheiden.

Beispiel 2.1 (Werkzeugvarianten) Bei der objektorientierten Analyse und dem objektorientierten Entwurf werden Klassendiagramm-Editoren eingesetzt. Die Darstellung sowie die änderbaren Eigenschaften von Klassen, Beziehungen, Attributen und Operationen unterscheiden sich geringfügig in den verschiedenen Phasen (ungerichtete vs. gerichtete Assoziationen, Angabe der Sichtbarkeit von Klassen, Attributen und Operationen, Signatur von Operationen). Außerdem müssen im Entwurf zusätzliche Konsistenzbedingungen geprüft werden und Kommandos etwa zum Generieren von Dokumentationen zur Verfügung stehen. In der anschließenden Implementierungsphase kann der Quelltext von Operationen bearbeitet und es können Klassengerüste in der gewünschten Programmiersprache generiert werden.

Um die Qualität des Entwurfs zu ermitteln, müssen Messungen ausgeführt und die Ergebnisse (falls nötig graphisch) dargestellt werden. Gleichzeitig benötigt der Projektmanager einen Überblick über die zu entwickelnden Pakete oder Klassen, ihren Fertigstellungsgrad, Aufwände und eventuelle Terminüberschreitungen – und damit die Werkzeuge, die die passenden Funktionen und Darstellungen anbieten.

□

Das Beispiel zeigt, daß hier verschiedene Werkzeuge benötigt werden. Die Eigenschaften und Funktionen dieser Werkzeuge ergeben sich zum einen aus der Phase oder Aufgabe, innerhalb derer sie eingesetzt werden (Analyse/Entwurf); zum anderen aus der Rolle, die ihr Benutzer

übernimmt (Entwickler/QS-Ingenieur).

Es ist nicht sinnvoll, mehrere Werkzeuge vollständig (und zum großen Teil redundant) zu spezifizieren und zu bauen. Bei der Nutzung eines Frameworks würden komplexe Vererbungshierarchien entstehen, in denen die Auswirkungen von Änderungen nur schwer nachvollziehbar sind. Der Meta-CASE-Ansatz sollte es daher ermöglichen, die unterschiedlichen *Werkzeugeigenschaften* zu spezifizieren und die Gemeinsamkeiten in den verschiedenen Varianten wiederzuverwenden.

In *MetaEdit+* können Teile der Methodendefinitionen (*method fragments*) als fertige Bausteine wiederverwendet werden. *Maestro II/Decamerone* unterstützt auch die Spezialisierung von Fragmenten durch eine *is a*-Beziehung [238].

Die verschiedenen Sichtweisen auf den Software-Prozeß und seine Produkte wurden im *Viewer*-Prototyp [258] berücksichtigt. Die verschiedenen Sichtweisen (*view points*) [260] der Prozeßteilnehmer werden separat modelliert. Jeder *view point* enthält die Beschreibung der zu verwendenden Notationen und der Vorgehensweise sowie (auf Instanzebene) die bearbeiteten Produkte, den aktuellen Prozeßzustand und die Prozeßgeschichte.

2.3 Prozeßorientierte Umgebungen

Prozeßorientierte Software-Entwicklungsumgebungen (PSEU) nutzen eine explizite Repräsentation des Software-Entwicklungsprozesses (das *Prozeßmodell*) zur Unterstützung der Benutzer bei der Ausführung solcher Prozesse. Nutzer einer PSEU sind Entwickler, die im Prozeßmodell vorgegebene Aufgaben ausführen, und technische Projektleiter (Manager), die den Prozeß planen, überwachen und steuern. Das Prozeßmodell soll die Kommunikation der Beteiligten erleichtern und dazu beitragen, daß die Beteiligten den Prozeß besser verstehen. In Tabelle 2.1 sind die Ziele aufgelistet, die eine Prozeßunterstützung in dieser Hinsicht verfolgt: Nur ein von allen Beteiligten verstandener und akzeptierter Prozeß kann sich positiv auf die Software-Entwicklung auswirken.

Ziele	**Prozeßverständnis und Kommunikation**
1	Den Prozeß in einer für den Menschen verständlichen Form darstellen
2	Den Austausch über den Prozeß und das Treffen von Entscheidungen erleichtern
3	Den Prozeß soweit formalisieren, daß die Beteiligten effektiv zusammenarbeiten können
4	Ausreichende Informationen zur Verfügung stellen, damit die Beteiligten den Prozeß ausführen können
5	Eine Basis für die Schulung der Beteiligten bilden

Tabelle 2.1: Ziele bzgl. Prozeßverständnis und Kommunikation (nach [162])

Eine große Zahl von PSEU ist entwickelt und beschrieben worden (z.B. in [8, 262]). Die Systeme unterscheiden sich in der *Prozeß-Modellierungssprache* (*Process Modeling Language*, PML), der Art der Datenverwaltung, der Unterstützung von Kooperation und Verteilung, sowie den Möglichkeiten zur Integration von Entwicklungswerkzeugen – um nur einige Aspekte zu nennen. Mit PSEU kann die Prozeßausführung automatisiert werden: Eine virtuelle Maschine (die *Prozeßmaschine*) überwacht und steuert den Prozeßfortschritt auf Basis des

Prozeßmodells, sorgt also dafür, daß die im Prozeßmodell definierten Aufgaben in der vorgegebenen Reihenfolge und unter Berücksichtigung von Abhängigkeiten und Randbedingungen ausgeführt werden. Die Prozeßmaschine reagiert dabei auf Rückmeldungen von Benutzern und Werkzeugen, die im Prozeß arbeiten. Die Ziele der Prozeßautomatisierung sind in Tabelle 2.2 zusammengestellt.

Ziele	Prozeßautomatisierung
1	Eine effektive Software-Entwicklungsumgebung definieren
2	Unterstützung, Hinweise und Referenzmaterial anbieten, die die Arbeit der Beteiligten erleichtern
3	Wiederverwendbare Prozeßbeschreibungen in einem Repository verwalten
4	Prozeßfragmente automatisieren
5	Die Kooperation zwischen einzelnen Beteiligten und Gruppen durch Automatisierung von Routinetätigkeiten unterstützen
6	Automatisch Meßwerte sammeln, die die Erfahrungen mit dem ausgeführten Prozeß widerspiegeln
7	Die Einhaltung von Regeln zur Sicherung der Integrität des Prozesses überwachen

Tabelle 2.2: Ziele bzgl. Prozeß-Automatisierung (nach [162])

Entscheidend ist, daß die Prozeßautomatisierung dazu beitragen kann, die Effizienz der Entwickler und die Qualität des Prozesses zu erhöhen – und damit auch die Qualität der Produkte.

Neben der eigentlichen Produktion von Software müssen PSEU auch das Management des Software-Prozesses mit Werkzeugen unterstützen. Tabelle 2.3 zeigt, welche Ziele dabei ver-

Ziele	Prozeß-Management
1	Einen projektspezifischen Prozeß entwickeln, der die Eigenschaften eines bestimmten Projekts berücksichtigt, wie Produkte oder organisatorische Randbedingungen
2	Über die Art, wie Software erzeugt und gewartet wird, nachdenken
3	Die Entwicklung von Projektplänen unterstützen (Voraussagen)
4	Den Prozeß überwachen, steuern und koordinieren
5	Eine Basis für Messungen bereitstellen, z.B. durch die Definition von Meßpunkten

Tabelle 2.3: Ziele bzgl. Prozeß-Management (nach [162])

folgt werden. Durch die Überwachung des laufenden Prozesses können Projekt-Manager Probleme früh erkennen und gegensteuern, außerdem kann die Organisation aus den Erfahrungen lernen. Die Bewertung von Produkten und Prozeß kann durch Messungen [228] unterstützt werden.

Probleme mit PSEU

Leider spielen PSEU in der praktischen Software-Entwicklung nur eine unbedeutende Rolle [123]. Mögliche Gründe für diese Situation sind:

– Die Prozesse, die der Software-Entwicklung zugrundeliegen, sind nicht ausreichend verstanden, um sie automatisieren zu können. Sich ändernde, falsche oder übersehene Anforderungen, zum Teil unbekannte Herstellungs- und Einsatzbedingungen und der hohe

Anteil 'kreativer' Tätigkeiten in der Software-Entwicklung bilden eine schlechte Grundlage für die exakte Beschreibung von Aufgaben und deren Abhängigkeiten. Mack [233] sieht daher größere Ähnlichkeiten zwischen einem Software-Entwicklungsprojekt und einer Expedition zur Erkundung fremder Gebiete, denn der Steuerung eines Produktionsprozesses auf Basis festgelegter Regelgrößen. Bereits mehr als zehn Jahre vorher charakterisierten Liu und Horowitz [227] Entwurfsprojekte als *"applying new technologies to develop new products in an uncertain environment"*.

- Es widerstrebt den Software-Entwicklern, wenn ihnen ein Automat die nächsten auszuführenden Arbeitsschritte vorschreibt. Während einige Aufgaben innerhalb des Software-Entwicklungsprozesses gut automatisierbar sind (z.B. Konfigurationsverwaltung, Tests), sind gerade die 'kreativen' Tätigkeiten wie Analyse und Entwurf nur schwer in die Schablone eines Prozeßmodells zu pressen: Jeder Experte benutzt sein eigenes Modell, das sich auf Basis seiner Erfahrungen gebildet hat [312]. Eine ausgeprägtere Führung und Kontrolle ist aber für Anfänger sinnvoll.
- In vielen Fällen ist es nicht gelungen, zu zeigen, daß im Verhältnis zwischen Kosten und Nutzen der Prozeßtechnologie der Nutzen überwiegt. Wenn Entwickler und Manager die Vorteile nicht erkennen, werden sie den (meist recht hohen) Aufwand zur Implementierung einer Prozeßunterstützung ablehnen. Dies gilt vielfach auch für CASE-Werkzeuge [173].

Entscheidender Faktor für die Einsetzbarkeit von PSEU ist die Möglichkeit, sie an die Anforderungen innerhalb der Organisation oder des Projekts anpassen zu können. Dies kann nur durch eine hohe Flexibilität der Umgebung erreicht werden, die auch die rasche Erstellung von Umgebungsprototypen ermöglichen muß.

Der durch die Prozeßunterstützung entstehende zusätzliche Aufwand für den Entwickler sollte möglichst gering sein. Es sollten also möglichst wenige zusätzliche Arbeitsschritte nötig und möglichst wenige zusätzliche Werkzeuge oder Benutzungsschnittstellen zu bedienen sein.

Das Kosten-Nutzen-Verhältnis sollte durch einen möglichst hohen erkennbaren Nutzen der PSEU verbessert werden: Durch Messungen [231] kann die Qualität der Produkte und die Transparenz für das Management erhöht werden, ein Arbeitskontext ermöglicht die schnellere Bearbeitung von Aufgaben, und durch die Automatisierung von Routineaufgaben spart der Entwickler Zeit. Wichtig ist auch, daß der Entwickler bei seiner Arbeit nicht zu sehr eingeschränkt wird, sondern, im Rahmen festzulegender Grenzen, seine Arbeitsabläufe selbständig festlegen kann. Statt Programmierer als Prozeduren oder Maschinen anzusehen [159], denen zu einem beliebigen Zeitpunkt eine Aufgabe übertragen werden kann, sollten lediglich (bezogen auf das Prozeßmodell) falsche Aktionen verhindert, ansonsten die Arbeit aber nicht unnötig eingeschränkt werden.

2.3.1 Aufgaben der Prozeßunterstützung

Koordination

Koordination wird in [234] als *"management of dependencies between tasks"* bezeichnet. Die Abhängigkeiten zwischen Aufgaben sind:

- *Gemeinsame Ressourcen* können zum einen Personen, Geräte, Software-Lizenzen oder Räume sein, die zur Ausführung einer Aufgabe (unter Umständen exklusiv) benötigt werden; zum anderen Produkte, die in den Aufgaben be- und verarbeitet werden. Im hier

betrachteten Kontext sind letztere meist Entwurfsdokumente. Um diese parallel bearbeiten zu können, müssen die Dokumente für alle Beteiligten zugreifbar sein (*data sharing*), gleichzeitig müssen parallele Zugriffe synchronisiert (*concurrency control*) und die Konsistenz der Daten (*consistency*) gesichert werden [315].

- *Erzeuger-/Verbraucher-Beziehungen*, die den Fluß von Dokumenten zwischen Aufgaben beschreiben und damit auch eine Bearbeitungsreihenfolge festlegen. Sie werden daher auch als Datenflußbeziehungen bezeichnet. Zur Modellierung können Datenflußdiagramme oder Petri-Netze verwendet werden.
- *Synchronisationsbeziehungen* schränken die möglichen Reihenfolgen bei der Ausführung von Aufgaben ein, regeln also den Kontrollfluß. Hierzu zählen Vorgänger- und Nachfolgerbeziehungen zwischen Aufgaben. Neben der Synchronisation von Beginn und Ende einer Aufgabe können auch *feingranulare Abhängigkeiten* [191] zwischen Aufgaben bestehen. Sie ergeben sich aus dem Zustand der beteiligten Aufgaben, also etwa aus der Tatsache, ob der Vorgänger ein Dokument erzeugt hat, das der Nachfolger weiterbearbeiten kann.
- Mit *Unteraufgaben-Beziehungen* können umfangreiche Aufgaben in kleinere Arbeitspakete zerlegt werden (*work breakdown structure*) [227]. Die Zerlegung kann bis zur Ebene von Werkzeugaufrufen verfeinert werden.

Kommunikation

In großen Projekten nimmt der Aufwand für die Kommunikation zwischen den Beteiligten stark zu [324]. Eine Unterstützung für asynchrone Kommunikation (etwa durch elektronische Post oder Aufgabenlisten) und synchrone Kommunikation (etwa durch Telekonferenzen oder Mehrbenutzer-Editoren) ist daher sinnvoll [26]. Die Prozeßunterstützung kann zum einen Kommunikationsmechanismen nutzen, um die Beteiligten über wichtige Ereignisse zu informieren. Zum anderen kann sie die Kommunikation zwischen den Beteiligten erleichtern, indem sie ihnen Informationen über die Beteiligten und ihre Beziehungen zu Aufgaben und Dokumenten zur Verfügung stellt. Die Kommunikation muß dazu explizit modelliert werden [9].

Kollaboration

Arbeiten mehrere Prozeßteilnehmer in einem Prozeß zusammen, muß jeder die Möglichkeit haben, auf gemeinsame Informationen zuzugreifen und diese zu verändern. Wichtig ist hier zum einen die Möglichkeit, passende Zugriffsrechte zu vergeben; zum anderen müssen die Betroffenen durch einen Notifizierungsmechanismus über relevante Änderungen informiert werden können (*change awareness* [324]).

Die Prozeßunterstützung, die Dokumente zuteilt, Zugriffsrechte setzt und Werkzeuge bereitstellt, muß dabei an die gewünschte Art der Kollaboration angepaßt werden können. Bei der Bearbeitung einer Analyse- oder Entwurfsspezifikation sind folgende Arten der Zusammenarbeit denkbar [188]:

- Mehrere Versionen des Dokuments werden parallel und unabhängig bearbeitet, danach die Ergebnisse zu einem Dokument gemischt.

– Das Dokument wird gleichzeitig und kooperativ von mehreren Entwicklern mit Hilfe von Mehrbenutzer-Editoren bearbeitet.
– Ein Entwickler erstellt oder bearbeitet das Dokument; seine Änderungen werden anschließend von mehreren Entwicklern begutachtet, akzeptiert oder die Überarbeitung beschlossen.
– Das Dokument wird basierend auf einem 'Gliederungsdokument' in mehrere Teile aufgeteilt, die separat und unabhängig bearbeitet werden. Beispiele sind eine Paketstruktur oder ein Klassendiagramm. Voraussetzung ist hier, daß Datenmodelle und Werkzeuge eine Aufteilung und getrennte Bearbeitung erlauben.

2.3.2 Prozeßmodellierungsparadigmen

Die meisten Ansätze zur Prozeßmodellierung benutzen mehrere Paradigmen, um Software-Entwicklungsprozesse zu beschreiben [13, 226]. Meist läßt sich aber ein zentrales Paradigma identifizieren; nach diesem sind einige bekannte Ansätze im folgenden klassifiziert. Die Auflistung soll dabei nur einen Überblick über die verschiedenen Ansätze vermitteln, ohne Anspruch auf Vollständigkeit zu erheben. Für eine ausführlichere Beschreibung sei z.B. auf [79, 262] verwiesen.

Netzbasierte Ansätze

Mit netzbasierten Ansätzen werden Software-Prozesse als Netz von Prozeßschritten modelliert, die durch Kontroll- und Datenflußbeziehungen verbunden sind. Hierzu zählen Netzpläne, Datenflußdiagramme und Petri-Netze. Letztere sind die Basis für *FUNSOFT* [141] und *SLANG* [19] und werden in *ProcessWeaver* [111] verwendet. Die Plätze repräsentieren hier typisierte Lager für Dokumente und Ressourcen; Transitionen modellieren Aufgaben oder Aktionen. Zur Instantiierung des Prozesses wird eine Kopie des Netzes erzeugt und mit Marken belegt (*active copy*). Die Änderung des Prozeßzustands entspricht dann dem Schalten von Transitionen, wobei Marken durch das Netz wandern. Problematisch ist die Handhabung von dynamischen Veränderungen des Prozesses, da die Netzstruktur zum Zeitpunkt der Instantiierung des Netzes festgelegt ist. Daher wird in einigen Ansätzen die Modifizierung des Netzes mit speziellen Transitionen oder übergeordneten Netzen ermöglicht.

In *DYNAMITE* (*Dynamic Task Nets*) [157] werden Software-Prozesse mit Hilfe von *Graphersetzungssystemen* beschrieben. Ein laufender Softwareprozeß besteht aus einer Menge von Aufgaben und deren Beziehungen, also einem *Aufgabennetz*. Mit *Graphtransformationen* [291] wird festgelegt, welche Änderungen an diesem Netz erlaubt sind. Eine Transformation des Aufgabennetzes spiegelt also die Änderung des Entwicklungsprozesses in der realen Welt wider. Der Ansatz berücksichtigt besonders die Dynamik von Software-Prozessen, da etwa auf die Änderung der Produktstruktur flexibel reagiert werden kann [323].

Regelbasierte Ansätze

In regelbasierten Ansätzen wird durch Regeln festgelegt, welche Bedingungen erfüllt sein müssen, damit eine bestimmte Aktion ausgeführt werden kann (Vorbedingungen), und welcher Zustand nach der Ausführung gelten soll (Nachbedingungen). Soll eine Aktion ausgeführt werden, deren Vorbedingung nicht erfüllt ist, wird ein *Plan* berechnet, dessen Ziel

es ist, die Vorbedingung der Aktion zu erfüllen. Der Plan existiert dabei nur als transiente Struktur innerhalb der Prozeßmaschine und kann nicht direkt manipuliert werden. Regelbasierte Ansätze wie *Marvel* [126] eignen sich eher für die Modellierung von Aufrufketten von Werkzeugen als für Software-Prozesse mit ihrer größeren Dynamik. Eine Ausnahme ist *EPOS* [252], wo der Benutzer die Möglichkeit hat, einen berechneten Plan von Hand zu manipulieren. Die berechneten Pläne können auch zur Simulation von Prozessen genutzt werden und zum Aufbau einer Wissenbasis dienen [243].

Zustandsbasierte Ansätze

In zustandsbasierten Ansätzen werden Software-Prozesse durch (kooperierende) Zustandsautomaten beschrieben. *Statecharts* [154] besitzen eine reiche Semantik und eignen sich zur Beschreibung aller Arten von zustandsbasierten Systemen. Mit der zugehörigen Entwurfsumgebung *STATEMATE* [155] kann Quelltext generiert werden, der den modellierten Zustandsautomaten implementiert. *Entity Process Models* [172] verknüpfen die Zustandsmodellierung mit der Datenmodellierung, indem Dokumenttypen Zustandsautomaten zugeordnet werden. Die verschiedenen Zustandsautomaten kommunizieren über Nachrichtenaustausch miteinander. In *ESCAPE* [193] und *SOCCA* [106] werden analog EER-Modelle und *statecharts* zur Prozeßmodellierung genutzt. Die Transitionen entsprechen dabei Operationsausführungen. In *APEL* [72] werden *aufgabenspezifische* Zustandsautomaten definiert – ein Produkt kann also, abhängig von der Aufgabe, in der es bearbeitet wird, unterschiedliche Zustände besitzen und unterschiedliche Zustandsübergänge durchlaufen. Der Zustand des Produkts in einer Aufgabe wird dabei hierarchisch durch die möglichen Zustände in den Unteraufgaben verfeinert.

Ereignisbasierte Ansätze

In ereignisbasierten Ansätzen lösen Zustandsänderungen Ereignisse aus. Trigger reagieren auf diese Ereignisse und führen Prozeduren aus. Trigger werden z.B. in *APPL/A* [299], *ADELE-TEMPO* [27] und *MVP-L* [230] verwendet. Mit Triggern wird das reaktive Verhalten des Software-Prozesses implementiert. Sie sind daher Konzepte auf eher niedriger Abstraktionsebene, deren Verwendung schnell zu unübersichtlichen Prozeßprogrammen führen kann.

Prozedurale Ansätze

Software-Prozesse werden als Programme in einer speziellen Programmiersprache implementiert. Prozeßschritte entsprechen dabei einzelnen Anweisungen oder Routinen. Es stehen Kontrollstrukturen wie Verzweigungen und Schleifen zur Verfügung. Bekanntester Ansatz ist *APPL/A* [299].

Agentenbasierte Ansätze

Agentenbasierte Ansätze zeichnen sich besonders durch ihre hohe Flexibilität aus, da die Logik zur Ausführung des Prozesses auf verschiedene, auch verteilt arbeitende Agenten verteilt ist. Ein Software-Agent ist nach Jennings *"a self-contained program capable of controlling its own decision making and acting, based on its perception of its environment, in pursuit of one or more objectives"* [186]. Er kann also eigene Entscheidungen treffen und muß dabei seine Umgebung oder Situation ebenso berücksichtigen wie das Ziel, das es zu erreichen gilt. Zur

Strukturierung können Agenten in *agencies* zusammengefaßt werden, in denen vorgegebene Delegationsstrukturen herrschen [183]. Entscheidungen zwischen gleichberechtigten Agenten werden im Rahmen von Verhandlungen (*negotiation*) getroffen.

Beim Einsatz von Agenten zur Prozeßsteuerung lassen sich folgende Ansätze unterscheiden [190]:

1. Bei einem *aufgabenbasierten Ansatz* werden zunächst Aufgaben, Aktoren und Rollen des Prozesses im Prozeßmodell beschrieben. Den einzelnen Aufgaben werden *reaktive Agenten* zugeordnet, die das Prozeßmodell während der Prozeßausführung interpretieren. Es wird also auf eine zentrale Prozeßmaschine verzichtet; der Ansatz ist somit dezentralisiert, modular und flexibel. Beispiele sind *SCALE* [266], wo *Process Model Agents* (auch alternative) Implementierungen von Aufgaben enthalten, und *MOKASSIN* [136]. Hier wird das Vorgehen bei der Prozeßausführung explizit durch ECA-Regeln beschrieben, die als Parameter für die Agenten dienen.
2. *Rollenbasierte autonome Agenten* werden für spezielle Aufgaben innerhalb des betrachteten Anwendungsgebiets entwickelt und nehmen unterschiedliche Rollen im Prozeß wahr. Die aus den Agenten gebildeten *agencies* spiegeln die Organisationsstruktur im gegebenen Anwendungsbereich wider [257]. Der Prozeß ist also gekapselt im anwendungsspezifischen Wissen der Agenten und in ihren Strategien zur Verhandlung und Interaktion mit anderen Agenten. Systeme wie *ADEPT* [185], *Redux* [269] und *PEACE+* [7] folgen diesem Ansatz.
3. *Mobile Agenten* ermöglichen die Migration von Prozessen zu verschiedenen Ausführungsstationen [241]. Der Agent ist für die Auswahl der geeigneten Station, die Migration eines Prozeßteils und die Verarbeitung der Ausführungsergebnisse verantwortlich. Eine solche Art der Prozeßausführung ist aber eher bei Geschäftsprozessen mit mehreren beteiligten Unternehmen sinnvoll.

Dem Vorteil der größeren Flexibilität bei der Prozeßdurchführung steht bei agentenbasierten Ansätzen der Nachteil gegenüber, daß die Prozeßbeschreibung (weil auf die verschiedenen Agenten verteilt) stärker fragmentiert ist. Dem wird im ersten Ansatz durch die aufgabenorientierte Beschreibung des Prozeßmodells entgegengewirkt, das zur Koordination der Agenten und zur deren Parametrisierung dient.

CSCW-Ansätze

Ansätze aus dem Bereich der *Computer Supported Cooperative Work* (CSCW) [137] konzentrieren sich eher auf die Frage, wie die Kooperation mehrerer Beteiligter innerhalb eines Projekts unterstützt werden kann, ohne Prozesse vorher explizit zu beschreiben [32]. CSCW-Systeme werden auch als Beispiele für *groupware*-Systeme betrachtet. *Groupware* wird definiert als *"Systeme, die auf Computern basieren und Gruppen von Personen bei der Ausführung einer gemeinsamen Aufgabe oder beim Erreichen eines gemeinsamen Ziels unterstützen und eine Schnittstelle zu einer gemeinsamen Umgebung bieten"* [95].

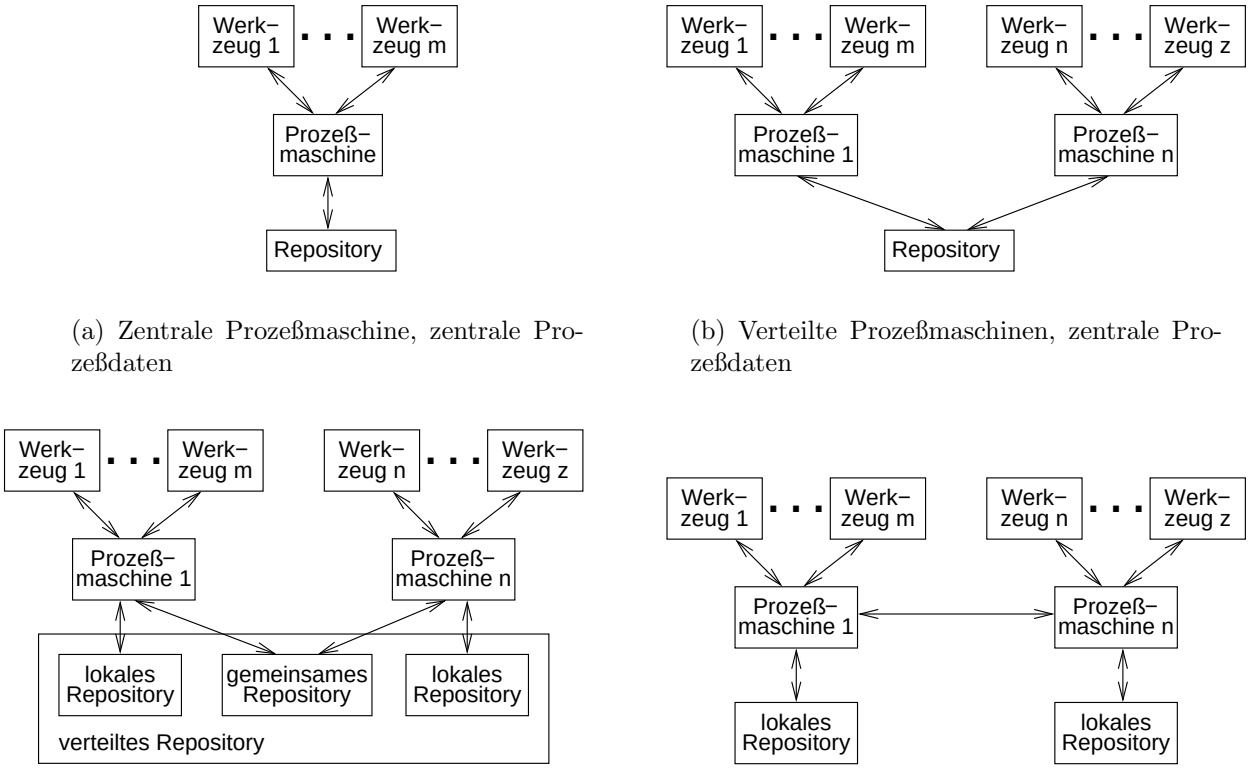
CSCW-Systeme beschränken sich zum Teil darauf, räumlich verteilten Projektteilnehmern den Zugriff auf gemeinsame Daten zu ermöglichen (wie in *BSCW* [10]). Andere bieten Sprachen zur Beschreibung von Aufgaben und ihren Abhängigkeiten (*ActionWorkflow* [240]) sowie Werkzeuge zu deren Durchführung und Überwachung an. Hierbei handelt es sich meist

um graphische Sprachen, da diese den Prozeß übersichtlich darstellen und von allen Beteiligten mit geringem Einarbeitungsaufwand verwendet werden können (z.B. *Visual Process Language*, *VPL* [300]).

In *Serendipity* [148] wird zur Prozeßmodellierung die Sprache *EVPL* (*Extended VPL*) verwendet, die VPL um Möglichkeiten zur Modellierung von Rollen, Produkten und Werkzeugen erweitert. Mit *VEPL* (*Visual Event Processing Language*) können vordefinierte Ereignisse an andere Aufgaben weitergeleitet werden. Beispiele für solche Ereignisse sind die Änderung des Aufgabenzustands oder die Änderung eines Produkts. Als Reaktion können auch benutzerdefinierte Aktionen ausgelöst werden.

2.3.3 PSEU-Architekturen

In [124] werden die folgenden Architekturen für PSEU vorgestellt. Bestandteile der PSEU sind CASE- und Prozeßwerkzeuge, die Prozeßmaschine und das Repository. Die Architekturen unterscheiden sich in der Verteilung der Prozeßmaschine und der Prozeßdaten (Abbildung 2.3).



(a) Zentrale Prozeßmaschine, zentrale Prozeßdaten

(b) Verteilte Prozeßmaschinen, zentrale Prozeßdaten

(c) Verteilte Prozeßmaschinen, teilweise verteilte Prozeßdaten

(d) Verteilte Prozeßmaschinen mit direkter Kommunikation, verteilte Prozeßdaten

Abbildung 2.3: PSEU-Architekturen anhand von Beispielinstanzen

1. In Architektur (a) wird eine zentrale Prozeßmaschine verwendet. Sie arbeitet auf einem zentralen Repository, das die Prozeßdaten verwaltet.

2. In Architektur (b) arbeiten mehrere Prozeßmaschinen parallel. Dadurch werden Geschwindigkeitsengpässe vermieden, wenn mehrere Benutzer/Werkzeuge parallel arbeiten. Dies ist die verbreitetste Architektur unter den PSEU-Prototypen.
3. In Architektur (c) sind auch die Prozeßdaten zum Teil verteilt: Neben dem *gemeinsamen Repository*, das gemeinsame Daten verwaltet und zum Austausch von Daten zwischen den verschiedenen Prozeßmaschinen dient, werden an den verschiedenen Standorten oder in verschiedenen Arbeitsgruppen *lokale Repositories* bereitgestellt. Hierdurch verringert sich der Kommunikationsaufwand, zusätzlich sind die Daten besser vor unbefugtem Zugriff geschützt, da sie im Verantwortungsbereich der jeweiligen Gruppe verwaltet werden.
4. Architektur (d) verzichtet auf das gemeinsame Repository. Prozeßdaten werden ausschließlich in lokalen Repositories verwaltet. Die Prozeßmaschinen können direkt Daten austauschen. Hierzu ist allerdings eine Kommunikationsinfrastruktur erforderlich.

Bei der Betrachtung der Architektur von PSEU müssen auch zwei weitere Aspekte geklärt werden:

– Die interne Architektur der Prozeßmaschine ist entscheidend für ihre Erweiterbarkeit und Adaptierbarkeit. Eine modulare Architektur ist einer monolithischen vorzuziehen, da sie die Möglichkeit zum Austausch einzelner Komponenten bietet.
– Die Software-Entwicklungswerkzeuge innerhalb der PSEU können das gleiche Repository wie die Prozeßmaschine oder andere Datenverwaltungssysteme nutzen. Beide Möglichkeiten werden im nächsten Absatz betrachtet.

2.3.4 Datenverwaltung

In PSEU werden neben den Produktdaten in Form von Software-Dokumenten auch Prozeßdaten verwaltet. Zu diesen gehören das Prozeßmodell und die aktuelle Prozeßinstanz, die den Zustand des laufenden Prozesses repräsentiert. Welche Prozeßdaten gespeichert werden, hängt natürlich vom Prozeßmodellierungsansatz und der Implementierung der Prozeßmaschine ab. Durch die Trennung des Prozeßzustands von den Mechanismen zu dessen Manipulation [161] wird eine größere Flexibilität bei der Anpassung und Erweiterung der Prozeßmodellierungskonzepte erreicht [214], was insbesondere beim Bau von PSEU-Prototypen wichtig ist [237].

Objektorientierte Datenbank-Managementsysteme (OODBMS) haben sich dabei als gut geeignet erwiesen für die Verwaltung der Daten einer PSEU [102] und werden auch als "*key component*" einer PSEU bezeichnet. Relationale DBMS unterstützen Software-Entwicklungswerkzeuge hingegen nur schlecht [33].

Die CASE-Werkzeuge, die Prozeßmaschine und die Management-Werkzeuge können ihre Daten im gleichen Repository verwalten oder verschiedene Datenverwaltungssysteme benutzen:

– Ein *gemeinsames Repository* erleichtert die Datenintegration verschiedener Werkzeuge und der Prozeßmaschine. Auch dokumentübergreifende Beziehungen können verwaltet und ihre Konsistenz gewährleistet werden [105]. Solche Beziehungen können zwischen verschiedenen Dokumenten (mit möglicherweise unterschiedlichem Typ) sowie zwischen

Produkt- und Prozeßdaten verlaufen. Außerdem wird die Architektur der PSEU vereinfacht, da nur ein Datenverwaltungssystem vorhanden ist. Auch können alle Werkzeuge und die Prozeßmaschine die gleichen Dienste des OMS nutzen, somit sind auch Werkzeugkomponenten, die auf diese Dienste aufsetzen, in allen Werkzeugen nutzbar. Im folgenden gehen wir von einem gemeinsamen Repository für Produkt- und Prozeßdaten aus.

– Der Einsatz *verschiedener Datenverwaltungssysteme* hat den Vorteil, daß vorhandene Werkzeuge unverändert genutzt werden können und jedes Werkzeug das optimale Datenverwaltungssystem verwenden kann. Die Werkzeuge sind dann aber nicht in die PSEU *integriert*: Sowohl die Steuerungsintegration (z.B. mittels *broadcast message server*) also auch die Datenintegration muß nachträglich realisiert werden – sofern das Werkzeug dies ermöglicht (s. auch Abschnitt 2.3.5).

  Ein (aufwendiger) Ansatz für die nachträgliche Integration von Werkzeugen wurde im *Desert Environment* [281] realisiert. Hier sorgt eine Datenbank für die Datenintegration: Sie speichert Verweise auf und Informationen über Dateifragmente. Dateifragmente sind z.B. Klassen, Methoden, Kapitel oder Absätze. Eine Komponente in *Desert* prüft periodisch, ob eine Datei geändert wurde und aktualisiert in diesem Fall die Datenbank. Der Benutzer arbeitet also mit einem 'logischen' Datenbestand, der aus beliebigen Fragmenten zusammengesetzt werden kann.

Aufgaben von OMS

Ein OMS kann in einer PSEU eine Vielzahl von Aufgaben erledigen und so zur Verringerung des Implementierungsaufwands ebenso wie zur Reduzierung der Komplexität der PSEU beitragen. Im folgenden werden einige wichtige OMS-Dienste erläutert.

Natürlich gelten für prozeßorientierte SEU die Erläuterungen über OMS-Dienste aus Abschnitt 2.1.2. Eine PSEU verwaltet aber neben den Software-Dokumenten weitere Daten: das Prozeßmodell, den aktuellen Prozeßzustand und die Prozeßgeschichte, sowie die Beziehungen zwischen Prozeß und Software-Dokumenten.

Verwaltung des Prozeßzustands. Der Zustand des Entwicklungsprozesses wird gebildet aus den Zuständen der einzelnen Aufgaben und der bearbeiteten Dokumente. Zusätzlich müssen die Beziehungen zwischen Aufgaben, Dokumenten und den Benutzern, die die Aufgaben ausführen, berücksichtigt werden. Das OMS muß also auch hier ein komplexes Datenmodell mit zahlreichen Konsistenzbedingungen verwalten und effiziente Operationen auf den Daten unterstützen. Durch Verwendung eines gemeinsamen Repositories für Produkt- und Prozeßdaten und eines integrierten Datenmodells können auch die Beziehungen zwischen Produkten und Prozeß direkt im OMS repräsentiert werden. In *MELMAC* [74] wird mit *GRAS* [211] eine Projektdatenbank realisiert, die alle relevanten Daten sowie die Beziehungen zwischen Aufgaben und Dokumenten enthält. Allerdings sind die Datenmodelle der Dokumente hier grobgranular – es können also keine Beziehungen zu einzelnen Dokumenteinträgen verwaltet werden. Das Datenmodell der Dokumente kennen nur externe Werkzeuge. Gleiches gilt für *DYNAMITE* [157]. Der Prozeßzustand wird hier als *Aufgabennetz* verwaltet, mögliche Editieroperationen werden durch Graphersetzungsregeln beschrieben.

Anders in *Goodstep*, wo Prozeß- und Produktdaten in O_2 verwaltet [102] und Werkzeuge zur Bearbeitung der feingranularen Dokumente aus *GTSL*-Spezifikationen generiert [97] werden.

In *Merlin* [194] hingegen wird der Prozeßzustand mit O_2 verwaltet, Werkzeuge arbeiten aber auf Dateien und einer *Gemstone*-Objektbank. In *Provence* [216] werden Prozeß- und Produktdaten separat verwaltet: Stellvertreter-Objekte (*proxies* [298]) im Prozeß-Repository enthalten die für die Prozeßausführung relevanten Informationen, eine Komponente namens *Enactor* ist für die Synchronisation von Produkt- und Prozeß-Repository zuständig [23]. Auf die Produkte wird nur über spezielle Werkzeuge zugegriffen.

Konsistenzbedingungen. In einer PSEU muß nicht nur die Konsistenz der Software-Dokumente gesichert werden, sondern auch die des Prozeßzustands. Konsistenzbedingungen gelten für [259]:

- *Beziehungen zwischen Aufgaben*: Die Aufgaben im Prozeß dürfen nur in einer bestimmten Reihenfolge ausgeführt werden; eine Aufgabe darf nur durch bestimmte Unteraufgaben verfeinert werden.
- *Beziehungen zwischen Aufgaben und Benutzern*: Es dürfen nur Benutzer mit bestimmten Rollen eine Aufgabe ausführen.
- *Beziehungen zwischen Aufgaben und Dokumenten*: Eine Aufgabe erwartet bestimmte Eingabedokumente und produziert bestimmte Arten von Ausgabedokumenten.
- *Zustände von Aufgaben*: Die möglichen Zustände einer Aufgabe können von den Zuständen anderer Aufgaben abhängen, etwa, wenn nur eine Aufgabe innerhalb eines Teilprozesses aktiv sein darf oder eine Aufgabe erst beendet werden darf, wenn andere Aufgaben abgeschlossen sind.
- *benutzerdefinierte Bedingungen*: Diese beziehen sich beispielsweise auf die einheitliche Verwendung von Begriffen oder die Berücksichtigung von Dokumentationsstandards.

Prinzipiell kann der Prozeßzustand als spezielles 'Dokument' betrachtet werden, so daß Mechanismen zur Konsistenzsicherung von Software-Dokumenten auch auf den Prozeßzustand anwendbar sind. Allerdings sind die Möglichkeiten, Konsistenzbedingungen im Datenbankschema auszudrücken, nur sehr begrenzt. Sie müssen daher durch Analysatoren oder Trigger (wie in *Goodstep* [100]) ergänzt werden, die komplexe Bedingungen prüfen können.

Zugriffsrechte. Die Verwaltung von Zugriffsrechten in feingranular modellierten Dokumenten ist schwierig, da an allen Dokumenteinträgen passende Rechte gesetzt werden müssen. Diese Aufgabe kann in PSEU erleichtert werden:

- *Definition von Zugriffsrechten*: Im Prozeßmodell ist festgelegt, welche Benutzer oder Rollen welche Rechte an welchen Dokumenten benötigen. So muß der Bearbeiter einer Aufgabe Schreibrechte an allen Ausgabedokumenten und Leserechte an allen Eingabedokumenten besitzen. Welche Konstellation von Zugriffsrechten an welchen (Arten von) Objekten gesetzt werden muß, kann also aus dem Prozeßmodell abgeleitet werden.
- *Setzen von Zugriffsrechten*: Die Prozeßmaschine kann automatisch Zugriffsrechte an Ressourcen setzen. Die Informationen hierzu erhält sie aus dem Prozeßmodell und dem aktuellen Prozeßzustand. Sie muß dazu überwachen, welche Dokument erzeugt wurden, in welchen Aufgaben sie bearbeitet werden und welche Benutzer als Bearbeiter den Aufgaben zugeordnet sind.

Zugriffsrechte sollten auch verwendet werden, um die Manipulation des Prozeßzustands durch die Benutzer der PSEU einzuschränken: Nur Manager dürfen Aufgaben erzeugen und Bearbeiter zuordnen. Letzteren kann das Recht eingeräumt werden, Unteraufgaben zu erzeugen und diese an andere Benutzer zu delegieren – sie übernehmen damit die Manager-Rolle für ihre Aufgaben.

Zugriffsrechte können allerdings nur sinnvoll genutzt werden, wenn die Werkzeuge diese auch berücksichtigen. Während bei grobgranular modellierten Dokumenten meist nur Schreib- oder Leserechte für das gesamte Dokument gewährt werden können, sind in PCTE verschiedene Zugriffsmodi für alle Komponenten eines Dokuments möglich. Die Werkzeuge müssen auf die jeweilige Rechtesituation sinnvoll reagieren, also nicht lesbare Einträge ausblenden oder Kommandos deaktivieren [206].

Sichten. Sichten spielen in PSEU sowohl auf der technischen, als auch auf der organisatorischen Ebene eine Rolle: Technisch benötigt jedes Werkzeug der PSEU eine bestimmte Sicht auf den gemeinsamen, konzeptionellen Datenbestand [242]. Unter organisatorischen Gesichtspunkten betrachten die verschiedenen Benutzer den Prozeß und seine Produkte aus unterschiedlichen Perspektiven, weil sie unterschiedliche Rollen übernommen haben oder unterschiedliche Aufgaben ausführen [313]. Mit Sichten können Informationen übersichtlich aufbereitet, gleichzeitig der Zugriff auf geschützte Informationen verhindert und Redundanz vermieden werden.

In PSEU besteht eine enge Beziehung zwischen Aufgaben, Rollen und Sichten: Die Kombination aus Aufgabe und Rolle bestimmt die passende Sicht; umgekehrt kann die Sicht auch als Konzept zur Kapselung von Aufgaben, Werkzeugen und Dokumenten verwendet werden. Die Sicht entspricht dann einem Standpunkt (*viewpoint* [260]), von dem aus der Software-Entwicklungsprozeß betrachtet wird [115]. Voraussetzung für die Nutzung von *viewpoints* sind feingranulare Prozeßmodelle [261]. *Viewpoints* können auch als Vehikel zur dezentralisierten Ausführung von Prozessen dienen [223]. Der modernere Begriff *processlet* [6] (ähnlich einem *applet* oder *servlet* in Java) deutet eher die Zerlegung des Prozesses in Teilprozesse an, die weitgehend autark ausgeführt werden und jeweils einen eigenen Zustand haben.

Sommerville et al. [295] betrachten Prozeßsichten eher aus dem Blickwinkel des Prozeßmodellierers, der vorhandene Prozesse analysieren und in geeignete Prozeßmodelle umsetzen muß: *Process viewpoints* erleichtern diesen Vorgang, da die verschiedenen Perspektiven separat betrachtet und verarbeitet werden können. Jeder *process viewpoint* wird durch ein oder mehrere Prozeßmodelle beschrieben, wozu auch unterschiedliche Formalismen genutzt werden können.

Avrilionis et al. geben in [16] eine formale Definition für Sichten. Sie zerlegen ein Petri-Netz in Subnetze, die die verschiedenen Sichten repräsentieren, und führen sie mit *ProcessWeaver* [111] aus. Sichten unterstützen hier auch die Weiterentwicklung des Prozesses, die Wiederverwendung von Prozeßfragmenten und die Beschreibung spezieller Aspekte wie die Berücksichtigung von Standards oder Qualitätskriterien. Letztere wurden im *MVP*-Projekt durch eine spezielle 'Qualitätssicht' repräsentiert [231].

In *ADELE-TEMPO* [27] werden Sichten als *Rollen* bezeichnet: Den Elementen im Prozeßmodell können verschiedene Rollen zugeordnet werden; die aktuelle Rolle legt die Eigenschaften

und das Verhalten des Elements fest. Das Rollenkonzept ist orthogonal zur Typhierarchie.

Das Prozeßmodell wird in *ALF* [47] in mehrere *MASP* (*Model for Assisted Software Process*) unterteilt. Jeder MASP enthält das Datenmodell, Operatortypen, ECA-Regeln und legt die Reihenfolge der Operatoraufrufe fest. Ein MASP modelliert damit alle Aspekte einer einzelnen Tätigkeit eines Benutzers.

Es liegt nahe, den Sichtenmechanismus des OMS zur Realisierung der Werkzeug- und Benutzersichten zu verwenden. Die Integration des Sichtenmechanismus mit der PSEU umfaßt

- das Ableiten der Sichtendefinitionen aus dem Prozeßmodell.
- das Ableiten der zu verwendenden Sicht für einen gegebenen Benutzer mit einer gegebenen Rolle aus dem aktuellen Prozeßzustand.
- die Erweiterung der Sicht um Informationen, die verfügbare Werkzeuge und ihre Eigenschaften bestimmen.

Lange Transaktionen. Die Transaktionen einer SEU unterscheiden sich von denen herkömmlicher betrieblicher Informationssysteme (vgl. Abschnitt 2.1.2, Seite 39). Entsprechend komplexere Transaktionsmodelle sind nötig, mit denen die Aufgaben im Entwicklungsprozeß und ihre Beziehungen beschrieben werden können.

In prozeßorientierten SEU wird der Prozeßablauf und die Kooperation der Entwickler im Prozeßmodell spezifiziert; die Prozeßmaschine sorgt dafür, daß bei der Prozeßausführung die Vorgaben des Prozeßmodells berücksichtigt werden.

Damit die Aufgaben im Entwicklungsprozeß einfach auf Transaktionen des OMS abgebildet werden können, müssen Transaktions- und Prozeßmodell möglichst ähnliche Eigenschaften aufweisen [218].

Einige PSEU basieren auf Konfigurationsverwaltungssystemen, z.B. *ADELE* [108] und *EPOS* [252]. Hier werden Aufgaben im Prozeß durch Entwurfstransaktionen realisiert. Den Entwurfstransaktionen sind Versionen der bearbeiteten Dokumente zugeordnet und es stehen Mechanismen zum Zurücksetzen und zur Synchronisation von Entwurfstransaktionen zur Verfügung. In *EPOS* werden die 'Absichten' von Transaktionen mit einer speziellen Sprache (*Transaction-Intention Definition Language*) [217] beschrieben, und so Konflikte bei der Ausführung vermieden. Die Planung der Transaktionen übernimmt ein *Transaction Planning Assistant* [61]: Aus dem Prozeßmodell werden Aufgaben, ihre Bearbeiter sowie Ein- und Ausgabedokumente ermittelt und Transaktionen so geplant, daß ein geringer Kommunikationsaufwand und wenig Konflikte entstehen. Letztere können aus den bearbeiteten Daten und dem 'Bestreben' (*ambition*) [64] einer Transaktion abgeleitet werden. In *Raleigh* [196] werden Aktionen im Prozeß mit Transaktionen gleichgesetzt. *Coordinated Activities* setzen Sperren auf den bearbeiteten Objekten, so daß die Isolation gewährleistet ist.

Trigger. Die Prozeßmaschine einer PSEU ist eine 'aktive' Komponente: Sie überwacht, welche Aktionen Entwickler und Manager ausführen, und reagiert darauf mit eigenen Aktionen. Der Trigger-Mechanismus eines OMS ist also gut nutzbar, um das reaktive Verhalten der Prozeßmaschine zu implementieren. So geschehen im *ALF*-Projekt [78], wo das PCTE-OMS um einen Trigger-Mechanismus erweitert wurde, und im *Goodstep*-Projekt, wo *active*

rules eingesetzt werden [59]: ECA-Regeln, deren Prädikate in O_2SQL und deren Aktionen in O_2C formuliert werden. In *ADELE* [108] sind Trigger Teil der Definition von Objekt- und Beziehungstypen und werden in der Typhierarchie vererbt. Sie dienen zur Konsistenzsicherung und zur Propagation von Änderungen zwischen den Arbeitsbereichen verschiedener Entwickler.

Der Nachteil von Triggern ist, daß die Implementierung der Prozeßmaschine auf mehrere Routinen verteilt wird, die noch dazu nach Regeln aufgerufen werden, die in den Triggern selbst festgelegt sind.

Versions- und Konfigurationsverwaltung. In Software-Prozessen besteht eine enge Beziehungen zwischen den Aufgaben und den Produktversionen, die in den Aufgaben bearbeitet werden: Bevor eine Aufgabe ausgeführt werden kann, muß zunächst festgelegt werden, von welcher Version der Eingabedokumente der Entwickler ausgehen soll. Das Ergebnis der Aufgabe sind wiederum neue Versionen der Ausgabedokumente. Eine Dokumentversion wird also auch durch die Aufgabe gekennzeichnet, in der sie erzeugt wurde [196] ('Klassendiagramm nach dritter Überarbeitung'). Die Konfigurationsverwaltung beschäftigt sich also mit der Frage, wie die Änderungen komplexer Produkte im Prozeß dargestellt werden können; mit Prozeßmodellen wird der Änderungsprozeß selbst beschrieben – eine Integration der Bereiche ist also sinnvoll [219].

In [264] wird auf die Identifizierung von Versionen ganz verzichtet und nur über Entwurfsaufgaben auf die Dokumente zugegriffen. *SUKITS* [322] hingegen verwaltet nur einen Konfigurationsgraph, der die Konfigurationen komplexer Produkte enthält. Jede Komponente in diesem Graph entspricht einer Aufgabe, in der die Komponente entwickelt wird. Damit der Prozeß tatsächlich ausgeführt werden kann, muß der Konfigurationsgraph um Informationen ergänzt werden, die zum Management des Prozesses gebraucht werden, etwa die Aufgabenzustände. In *DYNAMITE* [157] werden Software-Prozesse explizit modelliert und den Aufgaben externe Werkzeuge zugeordnet. Beim Starten eines Werkzeugs wird automatisch eine Version des Dokuments in den Arbeitsbereich des Entwicklers übertragen (*checkout*) und nachher die geänderte Version wieder zurückgestellt (*checkin*).

Andererseits ist es in PSEU auch sinnvoll, die Prozeßmodelle unter Versionskontrolle zu stellen [138]. So wird in *EPOS* die Versionsverwaltung der *EPOS-DB* auch für Prozeßmodelle genutzt [252]: Prozeßmodelle erleichtern die Konfigurationsverwaltung, die wiederum die Weiterentwicklung des Prozeßmodells unterstützt.

Abfragesprache. PSEU verwalten Daten mit einer komplexen Struktur. Eine Abfragesprache hilft dabei, Informationen aus dem Datenbestand zu ermitteln, abgeleitete Informationen zu erzeugen und die Daten geeignet aufzubereiten. Abfragesprachen können in Werkzeugen wie Analysatoren oder Reportwerkzeugen genutzt werden oder direkt von den Benutzern der PSEU zur Formulierung von *ad hoc*-Abfragen.

Im *Goodstep*-Projekt wird die O_2-Implementierung von *OQL* [3] genutzt. *Pleiades* [302] ist eine Erweiterung der Programmiersprache Ada um OMS-Funktionen. Sie bietet sowohl navigierenden als auch mengenorientierten Zugriff.

Verteilung. Große Software-Projekte werden häufig an räumlich verteilten Standorten ausgeführt [133, 325]. An den einzelnen Standorten müssen Entwickler und Manager einerseits autark arbeiten können, es müssen also alle relevanten Daten und Werkzeuge vorhanden sein und es muß ein Schutz vor unberechtigtem Zugriff von anderen Standorten aus existieren. Andererseits müssen die verschiedenen Standorte in einem gemeinsamen Prozeß kooperieren, also Daten austauschen und gemeinsam bearbeiten können. Da die PSEU die Aufgaben, ihre Bearbeiter und die zu bearbeitenden Dokumente kennt, kann die Prozeßmaschine für die Bereitstellung der Daten an den verschiedenen Standorten sorgen.

Ein Beispiel für eine solche *multi-site* PSEU ist *Oz* [31]: Die verschiedenen Standorte können jeweils eigene Prozesse definieren und die Kooperation mit anderen Standorten festlegen [30]. *CHIME* [87] unterstützt die verteilte Prozeßausführung mit Konzepten aus virtuellen Umgebungen, die die Navigation im Prozeß erleichtern sollen; ein ähnliches Ziel hat *LambdaMOO* [86]. Bei der Realisierung von verteilten PSEU kommen verstärkt Protokolle und Technologien des Internets zum Einsatz (z.B. [171, 325]).

Statt eine PSEU auf mehrere Standorte verteilt auszuführen, können auch mehrere PSEU im Verbund betrieben werden [306]: In diesem Fall muß ein 'Verbund-Prozeß' definiert werden, in den die verschiedenen möglicherweise unterschiedlichen PSEU eingebettet werden.

Schema-Evolution. Software-Projekte sind oft Änderungen unterworfen, die sowohl die Anforderungen und Ziele des Projekts, als auch die Art der Problemlösung und das Vorgehen betreffen. In einer PSEU können diese Änderungen Auswirkungen haben auf:

- das Prozeßmodell
- die Prozeßinstanz, sofern das geänderte Prozeßmodell gerade ausgeführt wird
- die Datenmodelle und Datenbankschemata [142]
- die verwendeten Werkzeuge

Das OMS sollte also die nachträgliche Änderung von Datenbankschemata unterstützen. Der Sichtenmechanismus des OMS erleichtert die Anpassung von Datenmodellen, da basierend auf dem konzeptuellen Schema neue Sichten definiert werden können, die die Änderungen reflektieren und die Integration neuer Werkzeuge erlauben.

Einige Datenverwaltungssysteme für PSEU

Einige PSEU-Ansätze stützen sich auf kommerzielle DBMS ab. Im *Goodstep*-Projekt [100] wurde O_2 [51] verwendet und um Sichten, Versionsverwaltung, Trigger und Schema-Evolution erweitert. Die *Groupie*-Umgebung [103], ebenfalls Teil des *Goodstep*-Projekts, verwendet *Gemstone* [45]. Feingranulare Datenstrukturen und Zugriffsoperationen werden hier in der Datendefinitionssprache *OPAL* beschrieben und Zugriffsrechte und Transaktionen zur Implementierung der Umgebungsfunktionen ausgenutzt.

ProSLCE [90] nutzt PCTE, wobei zur Datenmodellierung eine Teilmenge der Modellierungskonzepte aus dem PCTE-Standard verwendet wird. In *ALF* [78] und *COO* [48] wurden Erweiterungen von PCTE um Trigger, *workspaces*, *constraints* und *envelopes* zur Werkzeugintegration realisiert. Produkte werden grobgranular in *file*-Attributen gespeichert. Im Ansatz *PEACE* [12] wird die Datendefinitionssprache von PCTE zur Beschreibung des Objektmodells verwendet und den verschiedenen Prozeßfragmenten werden unterschiedliche Sichten

zugeordnet. Das Objektmodell wird ergänzt durch eine Beschreibung der Aktionen, die im Prozeß ausgeführt werden. Die Ein- und Ausgaben der Aktionen beziehen sich auf Objekttypen im Objektmodell. *LCPS* [80] nutzt die objektorientierte Erweiterung von PCTE [175] für die Definition und Ausführung von Prozessen – mit dem Ziel, eine standardisierte Basistechnologie für PSEU bereitzustellen. Der Ansatz schreibt dabei nicht vor, wie konkret die Aktionen im Prozeß beschrieben werden, kann also mit verschiedenen Formalismen genutzt werden. *P-Root* (*PCTE redesigned with object-oriented technology*) [48] definiert selbst eine objektorientierte Erweiterung von PCTE. Ziel des Projektes war es, eine Datenbank für SEU zu entwickeln. *COO* (*cooperation and coordination in the software process*) [131] nutzt *P-Root* und stellt Mechanismen zur Kooperation und Koordination in Software-Prozessen zur Verfügung [130].

Process Wall [161], ein OMS zur Verwaltung von Prozeßzuständen (*process state server*), basiert auf dem Objektspeicher *Triton* [160] und erweitert ihn um grundlegende Operationen zur Manipulation des Prozeßzustands. Ziel ist es, den Prozeßzustand vom Prozeßmodell und den Mechanismen zur Manipulation des Prozeßzustands zu trennen. Änderungen des Prozeßzustands werden mittels Nachrichten an die Prozeßmaschine propagiert. Auch *OIKOS* [251] hat das Ziel, standardisierte und anpaßbare Dienste für die Datenverwaltung in PSEU anzubieten, hierzu zählen auch Arbeitsbereiche und Versionierung.

In *MELMAC* [74] werden Projektdaten als Graphen in *GRAS* [211] gespeichert. Um den parallelen Zugriff mehrerer Komponenten der Umgebung zu ermöglichen, wird ein dedizierter Server-Prozeß gestartet, der allein auf das OMS zugreift und mit dem die Klienten über ein spezielles Protokoll kommunizieren. *GRAS* wird auch in *IPSEN* [212] verwendet. Die Zugriffsoperationen werden aus Graphersetzungsregeln (formuliert in *PROGRES* [291]) generiert. Analog nutzt *DYNAMITE* [157] *GRAS* zur Verwaltung von Aufgabennetzen, die den aktuellen Prozeßzustand repräsentieren. In *SUKITS* [322] speichert *GRAS* Versionsgraphen in technischen Entwurfsprozesses.

EPOS [252] und *ADELE-TEMPO* [27] basieren auf den Konfigurationsverwaltungssystemen *EPOS-DB* bzw. *ADELE*, die umfangreiche Konzepte für Versionierung und *workspaces* anbieten. Letztere befinden sich im Dateisystem. Werkzeuge können also unverändert auf den Dateien arbeiten. In der Datenbank wird der Versionsgraph verwaltet und Versionen mittels *checkin*- und *checkout*-Operationen zwischen Datenbank und *workspace* übertragen.

PRIME [275] verwendet ein relationales DBMS, ebenso *LEU* [82]. Datenbankschemata werden in *LEU* aus einem EER-Modell generiert. Allerdings steht hier die Unterstützung von Geschäftsprozessen mit einfach strukturierten Daten im Vordergrund.

2.3.5 Werkzeuge

Es gibt zwei Kategorien von Werkzeugen in einer PSEU: *Management-Werkzeuge*, mit denen Prozesse instantiiert, überwacht und gesteuert werden, sowie die eigentlichen *CASE-Werkzeuge*, die zur produktiven Arbeit der Entwickler dienen.

Management-Werkzeuge

Mit *Management-Werkzeugen* analysieren und überwachen Manager den Prozeßfortschritt oder greifen in den Ablauf ein. Wichtig ist hier, daß die Management-Werkzeuge den Prozeß

möglichst übersichtlich und leicht verständlich darstellen, andererseits auch den Zugriff auf Detailinformationen ermöglichen [323].

Zu den Management-Werkzeugen zählen auch Werkzeuge, die spezielle Benutzungsschnittstellen anbieten, mit denen Entwickler ihre Aufgabenlisten betrachten, Aufgaben ausführen und Rückmeldungen an den Prozeß schicken können. Diese Werkzeuge sollten zum einen möglichst nahtlos in die Produktionsumgebung des Entwicklers integriert, zum anderen an den jeweiligen Prozeß angepaßt sein.

In *LEU* [82] und *ADDD* [213] werden Benutzungsschnittstellen aus dem Prozeßmodell generiert. Auf die Produkte kann nur über geeignete Werkzeuge zugegriffen werden. Auch in *DYNAMITE* wird ein Generator-Ansatz verwendet [176]: Aus den Graphtransformationen, die das Prozeßmodell beschreiben, wird Quelltext für ein graphisches Werkzeug generiert. Mit dem Werkzeug wird dann eine Instanz des Prozeßmodells, also das Aufgabennetz, bearbeitet.

CASE-Werkzeuge

Mit CASE-Werkzeugen werden Dokumente bearbeitet, geprüft und transformiert. Die CASE-Werkzeuge werden von den Entwicklern genutzt und sollten mit der PSEU integriert sein. Bei der Bereitstellung und Integration von CASE-Werkzeugen in einer PSEU gibt es zwei Ansätze:

1. *Nachträgliche Integration vorhandener Werkzeuge*: PSEU bieten hierzu zwei Mechanismen an. Zum einen können Werkzeuge mit *envelopes* eingekapselt werden [311]. Diese bilden einen Adapter zwischen Werkzeug und PSEU. Beim Aufruf des Werkzeugs durch die PSEU übergibt der *envelope* die Aufrufparameter und liefert den Rückgabewert am Ende der Ausführung an die PSEU zurück. Eine Interaktion zwischen Werkzeug und PSEU ist also nur zu Beginn und am Ende der Werkzeugausführung möglich, was für nicht-interaktive Werkzeuge wie Compiler ausreicht [8]. *Provence* verwendet ein *smart file system*, das die PSEU über Dateiänderungen und Werkzeugausführungen informiert [216].

   Bei interaktiven Werkzeugen kommt eine nachrichten- oder dienstbasierte Integration [280] zum Einsatz, bei der Werkzeug und PSEU während der Werkzeugausführung kommunizieren. Das Werkzeug fragt bei der PSEU nach, welche Kommandos im aktuellen Prozeßzustand ausführbar sind. Führt der Benutzer ein Kommando aus, informiert das Werkzeug die PSEU. Die PSEU kann auch direkt Dienste des Werkzeugs aufrufen. Allerdings bieten die wenigsten CASE-Werkzeuge solche Schnittstellen an. Auch muß die Interaktion zwischen PSEU und Werkzeugen im Prozeßmodell auf sehr niedriger Abstraktionsebene implementiert werden und ist damit sehr fehleranfällig. Ein weiteres Problem ist die fehlende Datenintegration, da Werkzeuge und PSEU ihre Daten meist in unterschiedlichen Datenbanken oder Dateien speichern. Dies hat zur Folge, daß die Menge der Dokumente doppelt verwaltet werden muß – von den Werkzeugen und der PSEU. Beim Erzeugen und Löschen von Dokumenten müssen beide Mengen aktualisiert werden.

   In *SMART* [128] wurde die nachträgliche Integration kommerzieller Werkzeuge realisiert. Voraussetzung ist hier, daß die Werkzeuge zu *HP Softbench* [46] kompatibel sind: Prozesse werden zunächst mit *Articulator* [243] entworfen und simuliert, danach in *process templates* für *HP SynerVision* transformiert. *SynerVision* steuert die Prozeßausführung

und ist über einen *broadcast message server* mit einer *Softbench*-Instanz verbunden, in die wiederum die Werkzeuge integriert sind.

2. *Entwicklung neuer Werkzeuge*: Eine gute Prozeßintegration ist mit Werkzeugen zu erreichen, die unter diesem Gesichtspunkt entworfen wurden und damit für den Einsatz in einer PSEU ausgelegt sind. Ein Ansatz in dieser Richtung wurde im *Goodstep*-Projekt [100] entwickelt. Hier werden syntaxgesteuerte Werkzeuge für die verschiedenen Dokumenttypen eines Projekts mit der Werkzeugspezifikationssprache GTSL [97] beschrieben. Die Dienste der Werkzeuge werden ebenfalls in GTSL spezifiziert und können direkt von der Prozeßmaschine aufgerufen werden [96]. Ein anderer Ansatz ist PRIME [275], in dem Prozeß- und Werkzeugmodell bei der Konstruktion prozeßsensitiver Werkzeuge integriert werden.

Nur bei der Entwicklung neuer Werkzeuge ist auch die Daten- und Benutzungsschnittstellen-Integration der Werkzeuge mit der PSEU möglich. Datenintegration bedeutet hier, daß Beziehungen zwischen Prozeß- und Produktdaten verwaltet werden können, etwa indem einer Aufgabe die bearbeiteten Produkte zugeordnet oder indem Aufwands- und Qualitätsdaten gespeichert werden. Dazu müssen die Produktdaten hinreichend feingranular modelliert sein, so daß auch Teile eines Produkts wie einzelne Klassen innerhalb eines Diagramms identifiziert werden können.

Verwenden PSEU und Werkzeuge unterschiedliche Datenverwaltungssystem (z.B. hier eine Datenbank, dort Dateien) ist diese Integration, wenn überhaupt, nur umständlich möglich. Ein solcher Ansatz wurde in *Desert* [281] implementiert, wo in einer zentralen Datenbank Verweise auf Dateifragmente gespeichert und bei Änderung der Datei automatisch aktualisiert werden.

Die Schaffung einheitlicher Benutzungsschnittstellen unterschiedlicher Werkzeuge stellt im Zeitalter der graphischen Benutzungsschnittstellen mit den zugehörigen Gestaltungsrichtlinien [268] kein so großes Problem dar. Schwieriger ist die Erweiterung von CASE-Werkzeugen um prozeßspezifische Kommandos oder das Hinzufügen neuer Dialoge: Solche Erweiterungen werden nicht von allen Werkzeugen unterstützt; Werkzeuge unterschiedlicher Hersteller bieten i.a. auch unterschiedliche Schnittstellen und Verfahren zur Erweiterung an, so daß die Integration mehrerer Werkzeuge in einer Umgebung sehr aufwendig wird.

2.4 Prozeßunterstützung in Meta-CASE

Voraussetzung für den effizienten Einsatz von CASE-Werkzeugen ist die Unterstützung der verwendeten Methoden. Die Forschung im Bereich der Meta-CASE-Systeme konzentriert sich weitgehend auf den Produktaspekt der Methoden [214], also auf die Adaptierbarkeit von Werkzeugen an die verwendeten Dokumenttypen.

Nur wenige Arbeiten versuchen, eine Adaptierung der Werkzeuge an Konzepte, Notationen *und* Prozeßmodelle zu ermöglichen, so daß eine durchgängige Methodenunterstützung effizient konstruiert werden kann [214, 239]. Mi und Scacchi beschreiben in [244], wie sie eine CASE-Umgebung in einzelne Werkzeuge zerlegt und auf Basis eines Prozeßmodells erneut integriert haben. Dabei mußte auch das Datenmodell der CASE-Umgebung der Prozeßmaschine bekanntgemacht werden. Die Prozeßintegration beschränkt sich hier aber darauf, die

Werkzeuge in der richtigen Reihenfolge und mit den passenden Dokumenten aufzurufen.

Koskinen und Marttiin nutzen *MetaEdit+* zur Definition und Ausführungen von Software-Prozessen [215]: Ziel ist es dabei nicht nur, das Prozeßmodell an die jeweiligen Anforderungen anzupassen, sondern auch eine jeweils maßgeschneiderte Prozeß-Modellierungssprache (*"situational process modelling language"*) entwerfen, verwenden und weiterentwickeln zu können. Hierzu wurde das Meta-Modell um Prozeßkonzepte erweitert und es wurden Werkzeuge gebaut, mit denen PML entworfen und genutzt werden können. Allerdings greift die Prozeßmaschine hier nicht direkt auf die Dokumente zu und kennt ihr Datenmodell nicht. Informationen, die in den Dokumenten stecken, können also nicht für die Prozeßsteuerung genutzt werden.

Lyytinen et al. [232] integrieren die Meta-Modelle von *MetaEdit+* und *PRIME* [275], so daß Beziehungen zwischen den Aktionen im Prozeßmodell und den Konzepten im Dokument und ihren Darstellungen definiert werden können. Datenmodelle und Darstellungen der Werkzeuge werden also abhängig von der aktuellen Prozeßsituation zusammengesetzt. Der Entwurf von Werkzeugen auf diese Weise scheint aber sehr aufwendig, da sich sehr komplexe Werkzeugspezifikationen ergeben: Es ist nicht möglich, Werkzeugeigenschaften sinnvoll zusammenzufassen.

Nuseibeh und Finkelstein [260] kapseln die Produkte, Werkzeuge und Aufgaben eines Prozeßteilnehmers mit einer bestimmten Rolle in einem *viewpoint*. Ziel ist es, mit *feingranularen* Prozeßmodellen [223] eine gute Prozeßunterstützung für jeden einzelnen Entwickler zu erreichen. Allerdings liegt hier der Schwerpunkt auf der Definition und Auswahl von *viewpoints*; als CASE-Werkzeug wird nur ein einfacher Prototyp angeboten [258].

Grundy et al. [144] nutzen ein Framework und Meta-Werkzeuge für den Bau von CASE-Werkzeugen. Die Werkzeuge bieten eine Prozeßunterstützung, mit der die Aufgaben der Benutzer explizit beschrieben und ihnen Arbeitskontexte zur Verfügung gestellt werden können [146]. Werkzeuge und Prozeß werden über einen Mechanismus zur Änderungspropagation integriert, den das *JViews*-Framework zur Verfügung stellt: Werkzeuge senden Rückmeldungen über ausgeführte Aktionen, die zur Steuerung des Prozeßfortschritts genutzt werden. Auch Nachrichten über die Änderung von Dokumenten oder von Aufgabenzuständen werden auf diese Weise erzeugt und zur Aktivierung von Aufgaben oder zum Auslösen von Aktionen genutzt. Beim Bau der Werkzeuge werden Framework-Komponenten erweitert und angepaßt oder mit einem Meta-Werkzeug kombiniert. Das Framework ist sehr umfangreich, da es alle Dienste zur Datenverwaltung und Werkzeugintegration enthält. Beim Bau von Werkzeugen sind umfangreiche Kenntnisse des Frameworks nötig, da das Datenmodell, die verschiedenen Sichten der Werkzeuge und die Werkzeugeigenschaften mit Framework-Komponenten realisiert werden.

Anforderungen an PML

Folgende Eigenschaften sollte die verwendete PML aufweisen, um die Integration der Prozeßunterstützung in Meta-CASE-Umgebungen zu erleichtern [237]:

1. *Repräsentation des Prozeßmodells*: Um die Entwickler durch einen komplexen Prozeß führen zu können, sollte das Prozeßmodell zur Prozeßlaufzeit *explizit* repräsentiert sein, z.B. als graphisches Aufgabennetz. Bei einer *impliziten* Repräsentation kann der Benutzer

nur über Dialoge, Aufgabenlisten oder Menükommandos mit dem Prozeß interagieren, erhält also keine Übersicht über den Gesamtprozeß.

2. *Granularität des Prozeßmodells:* Bei der Verwendung von CASE-Werkzeugen unterscheidet Marttiin [237] drei Granularitätsstufen: (1) Aufgaben innerhalb des Software-Lebenszyklus (Analyse, Entwurf); (2) einzelne Modellierungsaufgaben (Erstellen eines Analyse-Modells) und (3) einzelne Schritte bei der Manipulation der Modelle (Erzeugen von Entitätstyp). Ebene (1) ist dabei zu grob, um einen Entwickler bei seinen konkreten Aufgaben zu unterstützen. Ebene (3) ist nur für ungeübte Benutzer sinnvoll, da Experten meist eigene 'Prozesse' zur Problemlösung entwickeln und eine Beeinflussung von außen eher ablehnen.
3. *Adaptierbarkeit des Prozeßmodells:* Der Entwurf von Software-Systemen kann wegen der zahlreichen Einflußfaktoren, die eine Änderung des Plans während der Projektdurchführung erforderlich machen, nur sehr eingeschränkt im voraus geplant werden. Daher ist die Möglichkeit zur dynamischen Adaptierung des Prozeßmodells an die jeweilige Projektsituation entscheidend für die Einsetzbarkeit des Ansatzes.
4. *Abstraktionsebene der Prozeßbeschreibung:* Nur wenn das Prozeßmodell den Entwicklungsprozeß auf hoher Abstraktionsebene beschreibt, kann es dazu beitragen, daß die Prozeßteilnehmer den Prozeß besser verstehen. Eine Beschreibung des Prozesses auf Implementierungsebene (wie z.B. in *APPL/A*) ist hier also nicht ausreichend. Sie muß durch eine Beschreibung der 'Prozeß-Architektur' ergänzt werden, die die Struktur des Prozesses verdeutlicht und auf die Darstellung von Details verzichtet.

2.5 PCTE und H-PCTE

PCTE (*Portable Common Tool Environment*) [174, 317] ist der ISO- und ECMA-Standard 13719 für Integrationsrahmen von SEU [202]. PCTE umfaßt zahlreiche Funktionen, darunter:

- Objektverwaltung
- Verwaltung von Betriebssystem-Prozessen und Interprozeßkommunikation
- Transaktionen
- Zugriffskontrollen
- Verteilung von Betriebssystem-Prozessen und Daten auf unterschiedlichen Rechnern
- Realisierung von graphischen Benutzungsschnittstellen

Alle Funktionen sind über eine im Standard definierte Schnittstelle zugreifbar. PCTE wird daher auch als *Portable* (oder *Public*) *Tool Interface* (PTI) [199] bezeichnet. Das PTI kapselt die darunterliegenden Betriebssystem-Dienste, so daß Werkzeuge leicht portierbar sind. Durch die Nutzung der Dienste des Integrationsrahmens wird die Integration der verschiedenen Werkzeuge einer SEU erleichtert und es werden Probleme wie die Duplizierung von Daten und Funktionen in verschiedenen Komponenten einer SEU vermieden [205].

H-PCTE [201] ist eine *h*auptspeicherbasierte und damit *h*ochperformante Implementierung großer Teile des Objektverwaltungssystems (*Object Management System*, OMS) von PCTE. H-PCTE bietet eine Programmierschnittstelle (API) für die Programmiersprache C. Über

das API wird *navigierend* auf die Objektbank zugegriffen, indem die Beziehungen zwischen Objekten verfolgt werden.

Viele der im folgenden beschriebenen Eigenschaften von H-PCTE können nur in Verbindung mit einer geeigneten Werkzeugarchitektur genutzt werden. Wie eine solche aussieht, beschreibt Abschnitt 3.1.

2.5.1 Datenbankmodell

Das Datenbankmodell von PCTE basiert auf dem ER-Modell. Es enthält Attribute, Objekt- und Beziehungstypen. Eine PCTE-Datenbank wird daher auch als *Objektbank* bezeichnet.

- *Objekttypen* definieren die Eigenschaften von Objekten. Ein Objekttyp besitzt eine Menge von Attributen und ausgehenden Beziehungen sowie eine (möglicherweise leere) Menge von Elterntypen (mehrfaches Erben).
- *Linktypen* definieren die Eigenschaften von Beziehungen (*Links*). Links treten nur als Paar von Hin- und Rücklink (Link und reverser Link) auf[1]. Ein Linktyp besitzt eine Menge von Zielobjekttypen. Bei Linktypen wird zwischen *Schlüsselattributen* und *Nicht-Schlüsselattributen* unterschieden. Ein Linktyp kann eine Liste von Schlüsselattributen (mit den Wertebereichen *natural* oder *string*) besitzen. Die Werte der Schlüsselattribute müssen je Objekt eindeutig sein.
- *Attribute* haben atomare Wertebereiche. Möglich sind die Wertebereiche *string*, *natural*, *integer*, *float*, *boolean*, *time* und *enumeration* (Aufzählung). Für Attribute kann ein Initialwert angegeben werden.

Die *Linkkategorie* legt semantische Eigenschaften von Linktypen fest. Die verschiedenen Linkkategorien im PCTE-Datenmodell sind in Tabelle 2.4 zusammengefaßt.

	Linkkategorie				
Semantische Eigenschaft	*composition*	*existence*	*reference*	*implicit*	*designation*
Komposition	+	–	–	–	–
Existenz	+	+	–	–	–
Referentielle Integrität	+	+	+	+	–
Relevanz für das Ausgangsobjekt	+	+	+	–	+

Tabelle 2.4: Linkkategorien

Die semantischen Eigenschaften haben folgende Bedeutungen:

- *Komposition*: Das Zielobjekt des Links ist Komponente des Ausgangsobjekts.
- *Existenz*: Der Link sichert die Existenz des Zielobjekts. Es muß also mindestens ein Link mit Existenz-Eigenschaft auf ein Objekt verweisen. Wird der letzte dieser Links gelöscht, wird auch das Objekt gelöscht.

[1] Ausgenommen sind hier Links der Kategorie *designation*: Sie besitzen keinen reversen Link, spielen aber an dieser Stelle keine Rolle.

– *Referentielle Integrität*: Es ist sichergestellt, daß das Zielobjekt des Links existiert. Über Links mit referentieller Integrität kann also immer navigiert werden. Umgekehrt gilt: Verweist mindestens ein Link mit der Eigenschaft der referentiellen Integrität auf ein Objekt, kann es nicht gelöscht werden.
– *Relevanz für das Ausgangsobjekt*: Bei einer Änderung des Links (erzeugen, löschen; ändern eines Attributwerts) gilt das Ausgangsobjekt als verändert.

Links der Kategorie *composition* werden im folgenden als *Composition-Links* bezeichnet; solche der Kategorie *reference* als *Reference-Links*. Links mit Kompositionseigenschaft spannen *komplexe Objekte* auf. Ein komplexes Objekt besteht aus einem Wurzelobjekt, allen über Composition-Links erreichbaren Objekten, sowie allen von diesen Objekten ausgehenden Links. Verweisen Composition-Links von mehreren Objekten auf ein anderes, wird dieses zur gemeinsamen Komponente (*shared component*). Für die Manipulation komplexer Objekte (kopieren, löschen, versionieren, verschieben) stehen spezielle API-Operationen zur Verfügung.

Zugriff auf die Objektbank

Eine Anwendung hält Verweise auf Objekte in der Objektbank. Diese werden als *Objektreferenzen* bezeichnet. Objektreferenzen werden durch *Navigation* durch die Objektbank erzeugt: Ausgehend von einem Objekt wird über einen *relativen Pfadausdruck* der Weg zu einem anderen Objekt beschrieben. Ein Pfadausdruck hat die Form

$$\textit{relativer Pfad} \quad ::= \quad \textit{Linkname} \;\; [\;\; \text{'/'} \quad \textit{Linkname} \;]\; * $$

mit

$$\textit{Linkname} \quad ::= \quad [\; \textit{Schlüsselattribute} \;] \quad \text{'.'} \quad \textit{Linktyp}$$

Besitzt der Linktyp mindestens ein Schlüsselattribut, so werden diese in der Form

$$\textit{Schlüsselattribute} \quad ::= \quad \textit{Schlüsselattribut} \quad [\; \text{'.'} \quad \textit{Schlüsselattribut} \;]\; *$$

angegeben.

Zum Einstieg in die Objektbank kann ein *Referenzobjekt* direkt adressiert werden. Hier relevante Referenzobjekte sind das Wurzelobjekt der Objektbank (*common root*, bezeichnet als '_'), und das *home*-Objekt eines Benutzers ('~'). Ein *absoluter Pfadausdruck* hat dann die Form

$$\textit{absoluter Pfad} \quad ::= \quad \textit{Referenzobjekt} \quad [\; \text{'/'} \quad \textit{relativer Pfad} \;]$$

Beispiel 2.2 (Pfadausdrücke)

– Absolute Navigation vom *home*-Objekt des Benutzers zu einer der Gruppen, in denen er Mitglied ist:
`~/0.home_of/1.user_member_of`
– Absolute Navigation vom Wurzelobjekt der Objektbank zum ersten *PI-SET*-Projekt:
`_/.piset/1.projects`
– Relative Navigation vom Dokumentobjekt zum ersten Eintrag eines ER-Diagramms:
`1.component_of_era_document`

□

Neben dem navigierenden Zugriff auf die Objektbank stehen in H-PCTE zwei weitere Zugriffsarten zur Verfügung [167]:

1. Der mengenorientierte Zugriff mit der SQL-/OQL-artigen Abfragesprache *P-OQL* [164] und der Abfragesprache *Ntt* [152]. Letztere berücksichtigt besonders *vage* Abfrageergebnisse, die entstehen, wenn Teile der Datenbank nicht zugreifbar sind [153].
2. Dokumenten-Retrieval [165], das beim Finden von Informationen in Freitexten wie Kommentaren und Beschreibungen hilft.

2.5.2 Schemata und Sichten

Ein Datenbankschema wird durch eine Menge von Typdefinitionen beschrieben und daher als *Schema Definition Set* (SDS) bezeichnet. In einem SDS können Typen *definiert* und Typen aus anderen SDS *importiert* werden. Ein Typ kann also in mehreren SDS vorkommen und in den verschiedenen SDS unterschiedliche Eigenschaften haben. Umgekehrt betrachtet kann die Definition verschiedener Merkmale eines Typs auf verschiedene SDS verteilt werden.

SDS werden in H-PCTE mit einer textuellen Datendefinitionssprache beschrieben und mit einem Schema-Compiler in die Meta-Datenbank übersetzt. Abbildung 2.4 zeigt einen Ausschnitt aus einer Schemadefinition. Die graphische Notation aus Abbildung 2.5 dient der Veranschaulichung von SDS und enthält nur einen Teil der Informationen aus der textuellen Definition.

Meta-Datenbank

Typen werden in PCTE *selbstreferentiell* durch Objekte und Links in der *Meta-Datenbank* verwaltet, einem speziellen Teil der Objektbank. Für die Manipulation der Meta-Datenbank werden spezielle API-Operationen verwendet.

Die Meta-Datenbank hat zwei Ebenen (Abbildung 2.6): Auf der *Type-Ebene* werden *globale* Eigenschaften von Typen verwaltet. Die *Type in SDS*-Ebene enthält *sichtenspezifische* Eigenschaften (vgl. [203]). Tabelle 2.5 gibt eine Übersicht über die für diese Arbeit relevanten Typeigenschaften.

Typ	**globale Typeigenschaften**	**lokale Typeigenschaften**
Objekttyp	Subtypen, Supertypen	Attribute, ankommende und ausgehende Linktypen
Linktyp	Kategorie, Schlüsselattribute, Umkehrlinktyp	Nicht-Schlüsselattribute, Ausgangsobjekttypen, Zielobjekttypen
Attribut	Wertebereich, Initialwert	—

Tabelle 2.5: Globale und lokale Typeigenschaften

```
-- SDS für ER-Diagramm
sds era:

-- Import von Typen aus SDS system, objectmodel, documents
import objecttype system-object;
import attribute system-system_key;
import attribute objectmodel-number;
import attribute objectmodel-specification;
import attribute objectmodel-position;
import objecttype documents-era_document;

-- Erweiterung von importiertem Objekttyp
extend objecttype era_document
with
   component component_of_era_document : composition link (number) to
             entitytype, relationshiptype, attribute
             reverse to_document;
end era_document;

-- Erweiterung von importiertem Linktyp
extend linktype component_of_era_document
with
   attribute position;
end component_of_era_document;

-- Definition von Objekttypen in Vererbungshierarchie
objecttype attributed_entity : child type of named_entity
with
   link to_attribute : reference link (number) to attribute;
end attributed_entity;

objecttype entitytype : child type of attributed_entity
with
   attribute weak : boolean;
   link is_a : reference link (number) to entitytype;
end entitytype;
...
end era;
```

Abbildung 2.4: Ausschnitt aus der SDS-Definition für ein ER-Diagramm

Typrechte

Die Typen im SDS können auch mit *Typrechten* (*type modes*) versehen werden. Typrechte schränken die Möglichkeiten zur Manipulation von Instanzen ein. Tabelle 2.6 enthält die *type modes* und ihre Bedeutung.

Arbeitsschema

Die aktuelle Sicht eines Werkzeugs wird als *Arbeitsschema* bezeichnet. Es wird durch eine Folge von SDS definiert und enthält die Vereinigung aller Typen aus den angegebenen SDS. Das Arbeitsschema dient also als Filter, durch den nur die Instanzen bestimmter Typen sichtbar sind (Abbildung 2.7). Das Arbeitsschema kann über spezielle API-Operationen abgefragt werden.

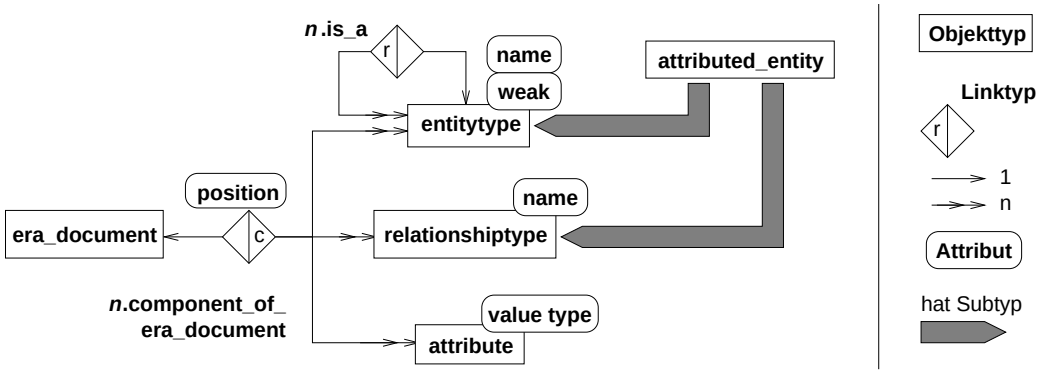
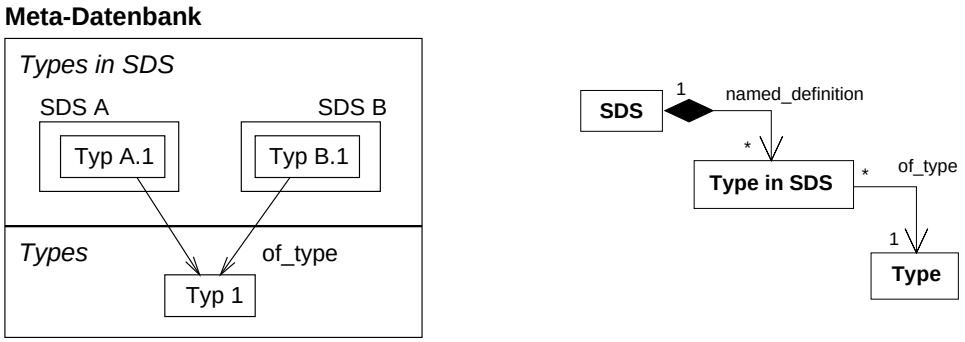


Abbildung 2.5: Graphische Notation für SDS



(a) Ebenen der Meta-Datenbank

(b) Meta-Schema

Abbildung 2.6: *Type in SDS*- vs. *Type*-Ebene in der Meta-Datenbank

2.5.3 Gruppenorientierte Zugriffskontrollen

Die Benutzer einer PCTE-Objektbank sind in *hierarchischen Benutzergruppen* organisiert. Die *Gruppen-Datenbank* enthält einen gerichteten azyklischen Graphen, dessen Knoten Benutzer und Benutzergruppen sind.

Die Kanten im Graphen verbinden zum einen Gruppen mit ihren Subgruppen, zum anderen Benutzer mit den Gruppen, in denen sie Mitglied sind. Abbildung 2.8 zeigt das Datenmodell der Gruppendatenbank und ein Beispiel.

Nach Kelter [200] werden die Zugriffsrechte durch ein 4-Tupel (S, G, M, V) festgelegt mit:

Type Mode	**gültig für**	**Bedeutung**
read, write	Attribute	Attributwerte von Objekten und Links können gelesen bzw. geschrieben werden
create, delete	Objekt-, Linktypen	Objekte und Links können erzeugt bzw. gelöscht werden
navigate	Linktypen	über die Links kann navigiert werden

Tabelle 2.6: *Type Modes* und ihre Bedeutungen

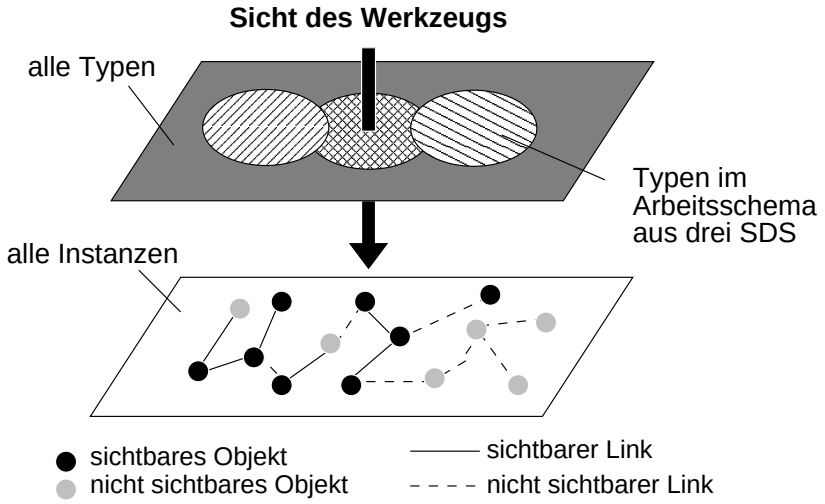


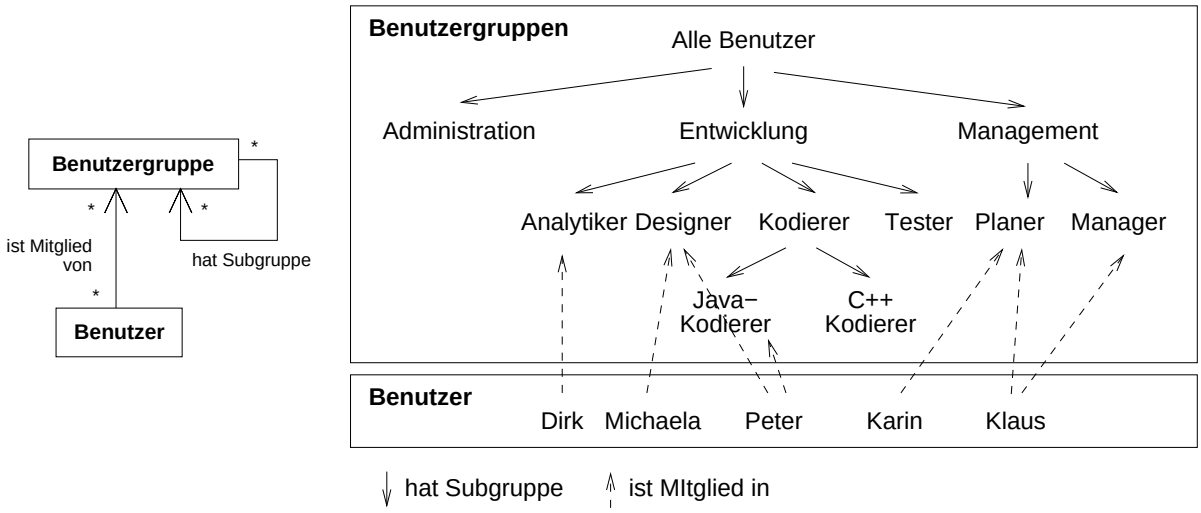
Abbildung 2.7: SDS und Arbeitsschema (aus [317])

- S: Das *Subjekt* (Benutzer, Benutzergruppe), dem die Rechte zugeordnet werden (Programme und Geräte werden hier nicht als Subjekte betrachtet). Ein Benutzer kann Mitglied in beliebig vielen Gruppen sein, hat aber zu jedem Zeitpunkt genau eine *aktuelle* Gruppe (*adopted group*). Die Rechte eines Benutzers ergeben sich aus den ihm selbst, seiner aktuellen Gruppe sowie deren Obergruppen zugeordneten Rechten.
- G: Das *Granulat* in der Objektbank, auf das zugegriffen werden soll: Hier atomare und komplexe Objekte.
- M: Der *Modus*, in dem zugegriffen werden soll, oder die Operationen, die auf dem Granulat ausgeführt werden sollen. In H-PCTE stehen 23 Zugriffsmodi zur Verfügung. Tabelle 2.7 enthält die hier relevanten Modi.
- V: Ein Wert, der bestimmt, ob der Zugriff erlaubt ist. Genauer sind in H-PCTE drei Werte für V möglich, `+`, `-` und `?`: Der Wert `+` erlaubt den Zugriff explizit, `-` verbietet ihn. `?` bedeutet, daß der Wert von V undefiniert ist.

Ob ein Benutzer mit einem Werkzeug eine Operation auf einem bestimmten Objekt ausführen kann, hängt ab von den Rechten, die ihm selbst, seiner aktuellen Gruppe und deren Obergruppen zugeordnet sind: Der Zugriff ist erlaubt, genau dann wenn er dem Benutzer selbst, seiner aktuellen Gruppe oder einer ihrer Obergruppen explizit erlaubt (`+`) und weder ihm noch einer der Gruppen explizit verboten (`-`) ist.

Die Gruppendatenbank in Abbildung 2.8 folgt dem *Rechtepaket-Paradigma* [198]: Eine Gruppe entspricht einem 'Paket' von Rechten, die allen Mitgliedern zugewiesen werden. Untergruppen haben weniger und spezialisiertere Mitglieder als die Obergruppen. Die Mitglieder der Untergruppe besitzen daher mehr Rechte als die Mitglieder der Obergruppe. Weiterhin ist zu bemerken, daß die Gruppenstruktur ein Projekt *qualitativ* in die verschiedenen Rollen oder Aufgabenbereiche (Entwicklung, Management, Administration) zerlegt. Alternativ könnte das Projekt *quantitativ* in Teilprojekte oder zu entwickelnde Subsysteme zerlegt werden.

In PCTE werden die Zugriffsrechte in einer *Access Control List* (ACL) am Objekt (G) gespeichert. Ein Eintrag enthält den Benutzer oder die Benutzergruppe (S) und die erlaubten und verbotenen Zugriffsmodi (M, V). Werden die Rechte nach dem Rechtepaket-Paradigma



(a) Datenmodell (b) Beispiel für eine Benutzer- und Gruppendatenbank

Abbildung 2.8: Gruppendatenbank

vergeben, enthalten die Einträge nur *erlaubte* Zugriffsmodi ($V = +$), deren Zahl beim Hinabsteigen in der Gruppenstruktur zunimmt.

Beispiel 2.3 (ACL) Eine mögliche ACL eines Analysedokuments zeigt die folgende Tabelle. Grundlage ist die Gruppendatenbank aus Abbildung 2.8(b).

Benutzer/-gruppe	**erlaubte Zugriffe**
Entwicklung	lesender Zugriff
Dirk	schreibender Zugriff, Ändern von ausgehenden Links, Eigentümerrecht
Manager	Löschen des Objekts

Alle Entwickler haben lesenden Zugriff; nur der Benutzer Dirk kann die Attribute des Dokuments schreiben und Dokumenteinträge erzeugen und löschen. Analytiker werden nicht explizit aufgeführt (also $V = ?$). Ihre Rechte ergeben sich aus den Rechten der Obergruppe Entwicklung. Alle Mitglieder der Manager-Gruppe können das Dokument löschen.

□

In H-PCTE sind nur atomare Zugriffsrechte implementiert. Um die Rechte an einem komplexen Objekt zu ändern, müssen die Rechte an allen Komponenten geändert werden. Die Werkzeuge sind für die Konsistenz der Zugriffsrechte innerhalb eines komplexen Objekts selbst verantwortlich.

2.5.4 H-PCTE-Prozesse

Zugriffe auf die Objektbank werden innerhalb von *H-PCTE-Prozessen* ausgeführt. Technisch sind H-PCTE-Prozesse als *leichtgewichtige Prozesse* (*threads*) realisiert [270]. Ein H-PCTE-Prozeß ist einerseits ein eigenständiger Kontrollfluß, und trägt andererseits Informationen

Zugriffsmodus	Bedeutung
Lesender Zugriff	
NAVIGATE	Navigieren über ausgehenden Link
READ_ATTRIBUTES	Lesen von Attributen
READ_LINK_ATTRIBUTES	Lesen von Attributen ausgehender Links
LIST_LINKS	Auflisten der ausgehenden Links
Schreibender Zugriff	
WRITE_ATTRIBUTES	Schreiben von Attributen
WRITE_LINK_ATTRIBUTES	Schreiben von Attributen ausgehender Links
Ändern der ausgehenden Links	
APPEND_LINKS	Hinzufügen von ausgehenden Links
DELETE_LINKS	Löschen von ausgehenden Links
Sonstige	
DELETE	Löschen des Objekts
CONTROL_DISCRETIONARY	Ändern der ACL am Objekt
OWNER	Benutzer (-gruppe) ist Eigentümer

Tabelle 2.7: Zugriffsmodi in PCTE und ihre Bedeutungen

wie:

- Das Arbeitsschema, also die externe Sicht des Prozesses.
- Den Benutzernamen und die aktuelle Gruppe des Benutzers. Der Benutzername und das Paßwort müssen beim Anmelden eines H-PCTE-Prozesses angegeben werden. Der Benutzer erhält eine voreingestellte aktuelle Gruppe, die er aber nachträglich ändern kann.
- Eine vorgegebene ACL, die an neu erzeugten Objekten gesetzt wird.
- Den Transaktionslog, mit dem Operationen rückgängig gemacht und wiederholt werden können.
- Vom Prozeß gehaltene Sperren.

In einer SEU, die auf der Objektbank arbeitet, können drei Arten von Prozessen unterschieden werden:

1. Ein (schwergewichtiger) *Betriebssystem-Prozeß*, der die gesamte Umgebung ausführt.
2. Mehrere (leichtgewichtige) *H-PCTE-Hauptprozesse*, die jeweils einen Funktionsbereich innerhalb der SEU ausführen (z.B. Administration, Entwurf, Projektmanagement).
3. Zu jedem H-PCTE-Hauptprozeß eine Hierarchie von *H-PCTE-Subprozessen*, die jeweils für verschiedene Werkzeuge/Fenster oder Kommandos im Werkzeug zuständig sind.

Ein Prozeß hält Sperren auf den von ihm bearbeiteten Ressourcen, sofern die Datenbank-Operationen innerhalb einer Transaktion ausgeführt wurden. Die Sperren verhindern, daß konkurrierende H-PCTE-Hauptprozesse gleiche Ressourcen ändern. Mehrere H-PCTE-Subprozesse eines Hauptprozesses werden hingegen nicht durch Sperren behindert: Sie können kooperativ auf gemeinsamen Ressourcen arbeiten. Mehr zu Transaktionen und Sperren in Abschnitt 2.5.7.

2.5.5 Verteilung und Segmentierung

In PCTE wird die Objektbank in logische Einheiten unterteilt, die *volumes* genannt werden. Verschiedene *volumes* können auf unterschiedlichen Rechnern im Netz liegen. Der PCTE-Standard enthält Vorkehrungen, um mit zeitweilig unzugreifbaren *volumes* umgehen zu können.

In H-PCTE wird die Objektbank in *Segmente* unterteilt. Alle Segmente müssen sich allerdings auf der gleichen Platte befinden. Da H-PCTE hauptspeicherbasiert arbeitet, wird ein Segment in den Hauptspeicher geladen, bevor ein Klient darauf zugreifen kann. Segmente können im H-PCTE-Server oder im H-PCTE-Klienten geladen werden:

- Im Server geladenen Segmente sind von allen Klienten zugreifbar. Beim Zugriff ist allerdings die Kommunikation zwischen Klienten- und Server-Prozeß nötig, wodurch die Zugriffsgeschwindigkeit deutlich verringert wird.
- In einem Klienten geladene Segmente sind nur für diesen zugreifbar, allerdings mit hoher Geschwindigkeit, da sich die Daten im Hauptspeicher des Klienten befinden (Benchmark-Ergebnisse sind in [270] zu finden).

Um eine hohe Zugriffsgeschwindigkeit und eine hohe Parallelität zu erreichen, müssen also möglichst zusammenhängende Bereiche der Objektbank, auf die jeweils nur ein Klient zugreifen möchte, auf einem Segment untergebracht werden.

In H-PCTE können Klienten auf verschiedenen Rechnern eines lokalen Netzwerks, die jeweils Zugriff auf die Segmentdateien haben, ausgeführt werden. Eine weite Verteilung von Klienten ist mit der Java-Schnittstelle möglich (Abschnitt 2.5.8).

2.5.6 Benachrichtigungsmechanismus

Arbeiten mehrere H-PCTE-Prozesse parallel auf der Objektbank, kann es passieren, daß Prozeß A eine Ressource ändert, deren Zustand Prozeß B zwischengespeichert hat. Ein Beispiel ist der Wert eines Attributs, den B als Text in einem Formular darstellt: Die Anzeige wäre nach der Änderung veraltet.

Um dies zu verhindern, enthält H-PCTE einen *verteilten Benachrichtigungsmechanismus* [271], der B über die Änderung informiert: B meldet *Notifizierer* an allen relevanten Ressourcen an (hier am angezeigten Attribut). Wird eine Ressource geändert, schickt H-PCTE eine Änderungsnachricht (*Notifizierungsnachricht*), die die Art der Änderung und, sofern sinnvoll, den neuen Zustand der Ressource enthält (hier den neuen Attributwert). Tabelle 2.8 zeigt die hier relevanten Notifizierungsnachrichten.

Beispiel 2.4 (Anmelden eines Notifizierers) ER-Diagramme werden mit dem SDS aus Abbildung 2.4 verwaltet. Um das Erzeugen von Einträgen in einem ER-Diagramm zu überwachen, wird beim Anmelden des Notifizierers angegeben: Die zu überwachende Ressource (eine Referenz auf das Dokument-Objekt), der Linktyp (component_of_era_document) und die zu überwachende Änderung (OBJECT_APPEND_LINK_OF_TYPE_EVENT).

Wird ein Link dieses Typs erzeugt, wird eine Rückruf-Funktion im Prozeß aufgerufen, die als Änderungsnachricht erhält: Die betroffene Ressource (Objektreferenz auf das Dokument),

Ereignis	Bedeutung
Ressource: ein bestimmtes Objekt	
OBJECT_MODIFY_EVENT	Wert eines Objektattributs geändert
OBJECT_APPEND_ANY_LINK_EVENT	Link an Objekt erzeugt
OBJECT_APPEND_VISIBLE_LINK_EVENT	Im Arbeitsschema sichtbarer Link erzeugt
OBJECT_APPEND_LINK_OF_TYPE_EVENT	Link eines bestimmten Typs erzeugt
OBJECT_DELETE_ANY_LINK_EVENT	Beliebiger ausgehender Link gelöscht
OBJECT_DELETE_VISIBLE_LINK_EVENT	Im Arbeitsschema sichtbarer Link gelöscht
OBJECT_DELETE_LINK_OF_TYPE_EVENT	Link eines bestimmten Typs gelöscht
OBJECT_DELETE_EVENT	Objekt gelöscht
OBJECT_MOVE_EVENT	Objekt verschoben
OBJECT_CONVERT_EVENT	Typ des Objekts konvertiert
OBJECT_MODIFY_ACL_EVENT	ACL des Objekts geändert
SET_LOCK_ON_OBJECT	Sperre an Objekt gesetzt
UNSET_LOCK_ON_OBJECT	Sperre an Objekt freigegeben
Ressource: ein bestimmter Link	
LINK_MODIFY_EVENT	Wert eines Linkattributs geändert
LINK_DELETE_EVENT	Link gelöscht
SET_LOCK_ON_LINK	Sperre an ausgehendem Link gesetzt
UNSET_LOCK_ON_LINK	Sperre an ausgehendem Link freigegeben
Ressource: ein bestimmtes Segment	
OBJECT_OF_TYPE_CREATE_EVENT	Objekt eines bestimmten Typs erzeugt
LINK_OF_TYPE_CREATE_EVENT	Link eines bestimmten Typs erzeugt

Tabelle 2.8: Notifizierungsereignisse und ihre Bedeutung

das Änderungsereignis und den neuen Zustand der Objektbank (Name des neuen Links). □

Da die Änderungsnachricht Informationen über den neuen Zustand der Objektbank enthält, ist meist kein weiterer Zugriff auf die Objektbank nötig, um den Werkzeugzustand zu aktualisieren.

Der Benachrichtigungsmechanismus kann eingesetzt werden

- *verteilt* zwischen verschiedenen schwergewichtigen Prozessen (verschiedene SEU-Ausführungen).
- innerhalb einer SEU-Ausführung zwischen verschiedenen leichtgewichtigen Prozessen (verschiedene Werkzeuge einer SEU-Ausführung).
- innerhalb eines leichtgewichtigen Prozesses: Da die Reaktion auf Notifizierungsereignisse ohnehin im Werkzeug implementiert werden muß, kann sie auch genutzt werden, um Ereignisse zu behandeln, die vom Werkzeug selbst ausgelöst wurden (*Selbstnotifizierung*). Ein Werkzeugkommando führt also nur die Änderung in der Objektbank aus (Erzeugen eines Dokumenteintrags); die Reaktion auf die Änderung (neuer Eintrag vorhanden) wird in der Rückruffunktion behandelt (neuen Eintrag in das Dokument einfügen und darstellen).

2.5.7 Transaktions- und Sperrmechanismus

Das *Werkzeug-Transaktionskonzept* von H-PCTE [270] unterstützt besonders die Realisierung von SEU, weil es folgende Eigenschaften berücksichtigt:

- Eine SEU-Ausführung enthält viele Werkzeuge, die mit unterschiedlichen Sichten auf der Datenbank arbeiten. Jedem Werkzeug wird also ein H-PCTE-Prozeß zugeordnet, der wiederum Subprozesse starten kann: Es entsteht eine Hierarchie von H-PCTE-Prozessen.
- Die Werkzeuge einer SEU-Ausführung müssen *kooperativ* arbeiten können, dürfen sich also nicht gegenseitig durch Sperren behindern.
- Werkzeuge verschiedener SEU-Ausführungen müssen bei Bedarf *isoliert* ausgeführt werden können, also geschriebene (und gelesene) Ressourcen für andere Werkzeuge sperren.
- In einer SEU werden sowohl interaktive Werkzeuge (Diagramm-Editoren, Browser) als auch nicht-interaktive Werkzeuge (Meß- und Prüfwerkzeuge, Transformatoren) eingesetzt. Während der Ausführung eines nicht-interaktiven Werkzeugs dürfen gelesene Ressourcen nicht verändert werden. Das nicht-interaktive Werkzeug kann dazu für die Dauer seiner Ausführung Ressourcen sperren und inkompatible Sperren anderer Werkzeuge *suspendieren*.

Jeder H-PCTE-Prozeß muß selbst eine Transaktion starten. Wird keine Transaktion gestartet, läuft der Prozeß im Modus *unprotected*: Es werden keine Sperren angefordert und Werkzeuge in verschiedenen SEU-Ausführungen können parallel und kooperativ auf gleichen Daten arbeiten. Ein Beispiel für diese Arbeitsweise sind *Mehrbenutzereditoren* [26], mit denen mehrere Benutzer gleichzeitig ein Dokument bearbeiten können. Ein Werkzeug kann also sowohl die geschützte als auch die kooperative Bearbeitung eines Dokuments erlauben. Die verschiedenen Arbeitsweisen sind dabei nicht im Werkzeug festgelegt, sondern werden allein durch den gewählten Transaktionstyp gesteuert – vorausgesetzt, das Werkzeug besitzt die in Kapitel 3 beschriebene Architektur.

Beim Starten einer Werkzeugtransaktion gibt der Prozeß den Transaktionstyp an. Von den in H-PCTE verfügbaren Transaktionstypen ist hier nur der Typ *Read Unprotected/Write Transactioned* (RUN/WTR) relevant – er kombiniert ungeschütztes Lesen mit impliziter Sperranforderung bei Schreibzugriffen: Parallel ausgeführte Prozesse können immer lesen; eine geänderte Ressource wird automatisch schreibgesperrt, kann also parallel nicht geschrieben werden. Das Problem der *dirty reads*, also das Lesen von Werten, die später von einem anderen Prozeß in der laufenden Transaktion geändert werden, kann mit dem Benachrichtigungsmechanismus umgangen werden.

Feingranulare Sperren

Sperren auf Ressourcen der Objektbank sind in H-PCTE *feingranular*: Es werden nur die tatsächlich betroffenen Ressourcen gesperrt. Wird in einer RUN/WTR-Transaktion das Attribut eines Objekts geändert, ist nur dieses für parallele Prozesse schreibgeschützt; andere Attribute und ausgehende Links können weiterhin geändert werden. Es treten also weniger Sperrkonflikte auf, wenn mehrere Werkzeuge parallel auf der Objektbank arbeiten; dadurch wird die Nebenläufigkeit erhöht.

Wichtig sind feingranulare Sperren auch in Verbindung mit Sichten: Sie gewährleisten, daß nur die Ressourcen von einem Prozeß gesperrt werden, die für ihn auch sichtbar sind –

durch die Beschränkung der Sicht eines Prozesses wird also auch die Parallelität bei der Bearbeitung erhöht.

Undo, Redo

Um einen konsistenten Zustand der Objektbank nach einem Systemfehler wieder herzustellen, werden alle Änderungen in einem *Transaktionslog* protokolliert. In H-PCTE wird dieser Log auch genutzt, um ein werkzeuggesteuertes Rückgängigmachen (*undo*) und Wiederholen (*redo*) von Operationen zu realisieren.

Das Werkzeug setzt dazu *Sicherungspunkte*, die zusammenhängende Operationsfolgen einklammern. Es gibt API-Operationen zum:

- Setzen von Sicherungspunkten. Ein Sicherungspunkt sollte nach jedem Werkzeugkommando gesetzt werden.
- Zurückspringen zu einem vorherigen Sicherungspunkt. Damit werden die letzten Benutzerkommandos rückgängig gemacht (*Undo-Funktion*).
- Vorwärtsspringen zum nächsten Sicherungspunkt. Damit werden rückgängig gemachte Änderungen wiederholt (*Redo-Funktion*), sofern der Prozeß seit dem letzten *undo* keine Änderungen gemacht hat.

2.5.8 Java-Schnittstelle

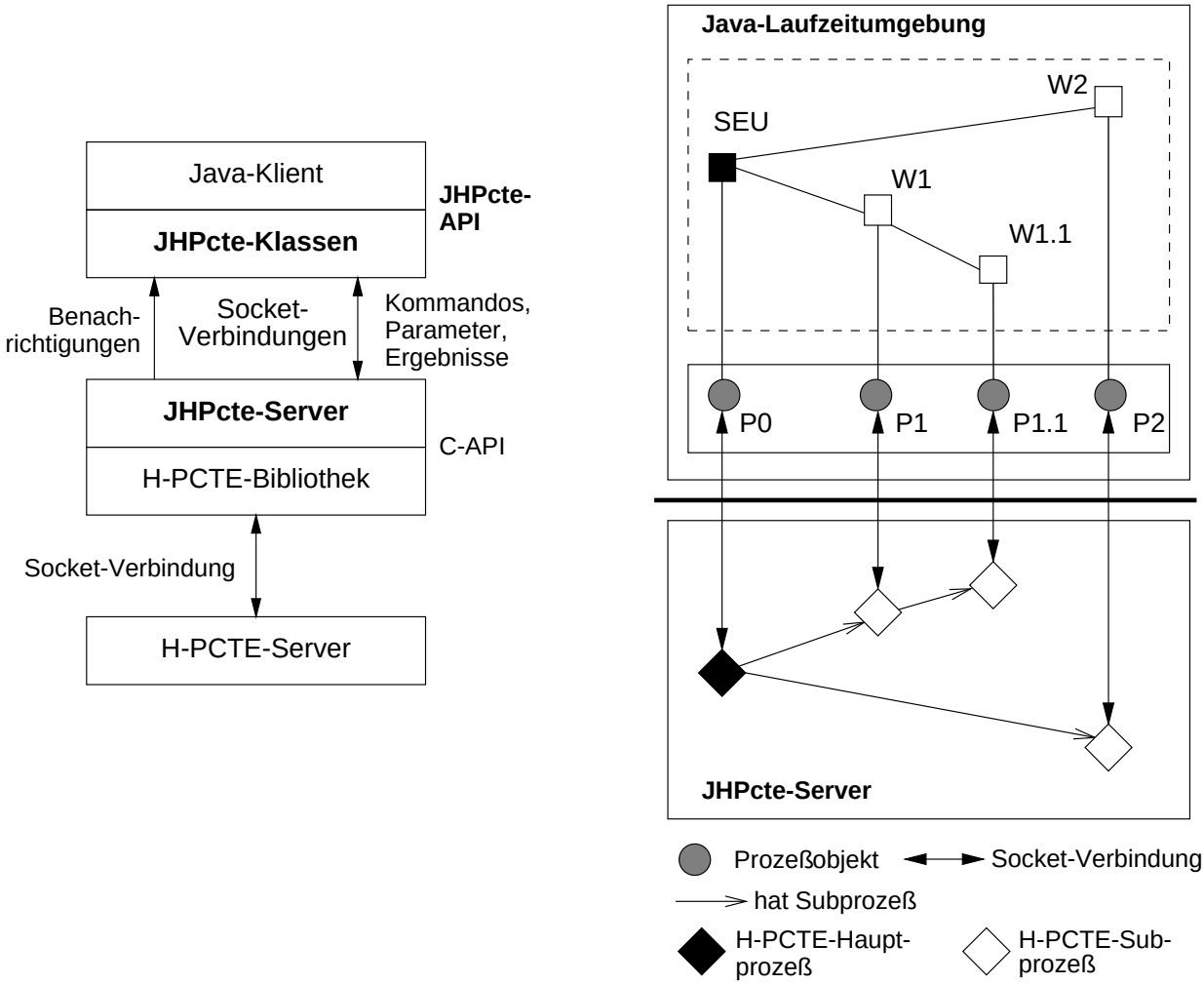
JHPcte [207, 208] ist die Java-Schnittstelle von H-PCTE. Sie ermöglicht es Java-Programmen, auf die Objektbank zuzugreifen. JHPcte besteht aus (Abbildung 2.9(a)):

- Java-Klassen, die das JHPcte-API implementieren. Ein Java-Klient erzeugt Instanzen dieser Klassen und kann über ihre Operationen auf die Objektbank zugreifen.
- Dem *JHPcte-Server*, der Kommandos vom Java-Klienten empfängt und in Operationsaufrufe des C-API von H-PCTE umsetzt. Java-Klient und JHPcte-Server kommunizieren über Socket-Verbindungen miteinander, können also auch auf unterschiedlichen Rechnern laufen.

Abbildung 2.9(b) zeigt, wie eine in Java implementierte SEU auf die Objektbank zugreift: Die gesamte SEU wird innerhalb einer Java-Laufzeitumgebung ausgeführt. Sie enthält mehrere Werkzeuge (W1, W1.1 und W2). W1 und W2 wurden direkt innerhalb der SEU gestartet (z.B. ein ER-Editor und ein Management-Werkzeug); W1.1 wurde von W1 gestartet (z.B. ein textueller Spezifikationseditor).

Der SEU selbst und jedem Werkzeug ist ein *Prozeßobjekt* zugeordnet. Das Prozeßobjekt repräsentiert einen H-PCTE-Prozeß im Java-Klienten, bietet Operationen zum Zugriff auf die Objektbank an und kapselt die Socket-Verbindungen zum JHPcte-Server.

Im JHPcte-Server ergibt sich bei der beschriebenen Konstellation von Werkzeugen eine *Hierarchie* von H-PCTE-Prozessen: Der SEU ist ein Hauptprozeß zugeordnet, der für W1 und W2 einen Subprozeß startet. Der Subprozeß für W1 startet einen weiteren Subprozeß für W1.1.



(a) Architektur (b) Instanzsicht

Abbildung 2.9: Zugriff eines Java-Klienten auf die Objektbank

Werden alle Zugriffe auf die Objektbank in Transaktionen ausgeführt, so können die Subprozesse der verschiedenen Werkzeuge innerhalb einer SEU kooperieren. Lediglich die Hauptprozesse verschiedener SEU (oder anderer Werkzeuge wie Browser und Administrationswerkzeuge) sind durch Sperren voneinander isoliert.

Kapitel 3

Dokumente und Werkzeuge

In diesem Kapitel werden zunächst die Grundlagen zum Werkzeugbau auf Basis des *object management system* H-PCTE vorgestellt. Wesentliche Merkmale sind die *OMS-orientierte Werkzeugarchitektur* und die Interpretation des Datenbankschemas durch die Werkzeuge. Daraus ergeben sich Anforderungen, die im Werkzeug-Konstruktionsansatz *genform* berücksichtigt werden: In *genform* werden Werkzeuge mit *Werkzeugschemata* spezifiziert (Abschnitt 3.2), die das Datenmodell und Parameter zur Steuerung der Werkzeuge enthalten. Die Werkzeuge werden dann aus *Werkzeugkomponenten* zusammengesetzt (Abschnitt 3.3), die direkt wiederverwendet oder werkzeugspezifisch erweitert werden können. Abschließend wird der Werkzeugbau mit *genform* anhand eines ausführlichen Beispiels in Abschnitt 3.4 erläutert.

3.1 Grundlagen, Geschichte, Ziele

Software-Entwicklungsumgebungen verwalten Daten mit komplexer Struktur. Ein *Software-Dokument* besteht aus *Einträgen* unterschiedlichen Typs. Der Typ eines Eintrags legt seine *Attribute*, *Komponenten* und ausgehenden *Beziehungen* fest. Beziehungen können sowohl innerhalb eines Dokuments verlaufen (*intra-Dokument-Beziehungen*), als auch zwischen verschiedenen Dokumenten (*inter-Dokument-Beziehungen*). Einträge können hierarchisch verfeinert werden – so entstehen *komplexe Einträge*. Ein komplexer Eintrag wird in manchen Fällen als Ganzes bearbeitet (gelöscht, kopiert); in anderen wird nur auf einzelne Komponenten zugegriffen. Ein Beispiel für einen komplexen Eintrag ist eine Klasse, die Attribute und Operationen mit Parametern enthält.

Jedes Werkzeug einer SEU arbeitet auf einer Teilmenge des gesamten Datenbestands. Auf der Typebene verfügen die verschiedenen Werkzeuge über unterschiedliche *Sichten* [242]: Die Sicht bestimmt, auf welche Daten ein Werkzeug zugreifen und welche es manipulieren kann. Wichtig ist, daß die unterschiedlichen Sichten konsistent bleiben [120], Änderungen auf gemeinsamen Daten sich also auf alle betroffenen Sichten auswirken.

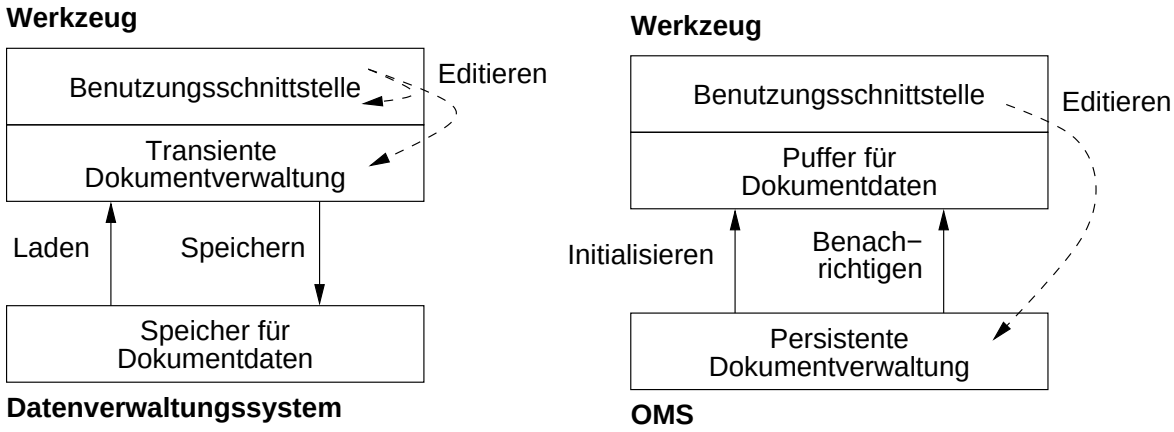
Ein Datenverwaltungssystem für SEU muß also einen Sichtenmechanismus, Zugriffskontrollen, Transaktionen, Versionierung und Dienste zur Konsistenzsicherung zur Verfügung stellen (s. auch Abschnitt 2.1.2). Die hohen Anforderungen, die SEU an Datenverwaltungssysteme stellen, wurden bereits früh erkannt [33] und *Repositories* [34] auf Basis objektorientier-

ter DBMS als aussichtsreiche Kandidaten vorgeschlagen [102, 105]. Aufgabe des Werkzeugentwicklers ist es, die OMS-Dienste in den Werkzeugfunktionen zu nutzen. Dadurch kann einerseits der Funktionsumfang der Werkzeuge erhöht, andererseits der Aufwand für die Implementierung der Werkzeuge verringert werden.

Die Forschung im Kontext von H-PCTE [201] hat zwei Ziele: Zum einen die Realisierung eines Repositories, das zahlreiche Dienste für SEU anbietet [152, 153, 164, 165, 166, 167, 205, 270, 271]; zum anderen die Entwicklung von Werkzeugarchitekturen, mit denen diese Dienste tatsächlich ausgenutzt werden können [68, 69, 204, 206, 207, 208, 246, 249]. Eine zentrale Anforderungen an die Werkzeugentwicklung ist dabei, daß Werkzeuge mit möglichst geringem Aufwand gebaut und einfach gewartet und erweitert werden können.

3.1.1 OMS-orientierte Werkzeugarchitektur

Werkzeuge einer SEU weisen üblicherweise eine Architektur wie in Abbildung 3.1(a) dargestellt auf: Beim Werkzeugstart oder nach einem Kommando wie 'Dokument öffnen' wird das Dokument vom persistenten Speicher in transiente Hauptspeicherstrukturen übertragen (*Laden*). Da das Dokument 'als Ganzes' geladen wird, ist es unerheblich, ob ein DBMS oder das Dateisystem zur persistenten Speicherung verwendet wird: Dokumente sind *grobgranular* modelliert – ihre interne Struktur ist also nur den Werkzeugen bekannt. Aufgrund der Syntax, der das Dokument genügt, kann ein Werkzeug eine transiente Struktur aufbauen.



(a) Traditionelle Werkzeugarchitektur

(b) OMS-orientierte Werkzeugarchitektur

Abbildung 3.1: Traditionelle vs. OMS-orientierte Werkzeugarchitektur

Über die Benutzungsschnittstelle (*graphical user interface*, GUI) des Werkzeugs kann der Benutzer das Dokument bearbeiten (*Editieren*). Er verändert dabei zum einen die transiente Kopie des Dokuments – das Original im persistenten Speicher ist also nach dem ersten Editierkommando veraltet. Zum anderen wird auch das GUI beeinflußt. Zur Aktualisierung des GUI kann die transiente Dokumentverwaltung lokale Änderungsnachrichten versenden. Erst beim *Speichern* des Dokuments wird die Kopie persistent gemacht.

Die transiente Dokumentverwaltung muß sowohl Datenstrukturen enthalten, um Einträge und Beziehungen in Dokumenten zu verwalten, als auch Kommandos, um Dokumente zu manipulieren. Hinzu kommen weitere Dienste wie Zugriffsrechte oder Maßnahmen für die Steuerungs- und Datenintegration [319] von Werkzeugen: In [14] wird der Anteil der transienten Dokumentverwaltung am gesamten Werkzeugquelltext auf 30 % geschätzt.

Anders bei einer *OMS-orientierten Werkzeugarchitektur* (Abbildung 3.1(b)): Sie verwendet ein OMS zur persistenten Datenverwaltung. Beim Start des Werkzeugs (oder später auf Anforderung) wird eine transiente Repräsentation des Dokuments *initialisiert*. Änderungen, die der Benutzer über das GUI vornimmt, werden hier direkt im OMS durchgeführt. Das OMS enthält also zu jedem Zeitpunkt die aktuellen Daten. Voraussetzung für das direkte Arbeiten auf der Objektbank ist, daß das Dokument *feingranular* modelliert ist, also die Feinstruktur des Dokuments dem OMS bekannt ist. Bei einer feingranularen Modellierung können Werkzeugkommandos wie das Erzeugen eines Eintrags oder das Ändern eines Attributs direkt in Operationen auf der Objektbank umgesetzt werden. Die transiente Repräsentation wird durch den Benachrichtigungsmechanismus des OMS aktualisiert (*Benachrichtigen*).

Die Vorteile der OMS-orientierten gegenüber der traditionellen Werkzeugarchitektur sind also:

- Da Werkzeugkommandos direkt im OMS durchgeführt werden, können Dienste des OMS wie Zugriffskontrollen und Transaktionen ausgenutzt werden; sie müssen also nicht erneut im Werkzeug implementiert werden.
- Die Objektbank enthält stets aktuelle Daten, so daß mehrere Werkzeuge kooperativ auf überlappenden Daten arbeiten können: Änderungen sind für alle Werkzeuge unmittelbar sichtbar; der Benachrichtigungsmechanismus des OMS sorgt für die Aktualisierung des Werkzeugzustands bei parallelen Änderungen.

Die transiente Dokumentverwaltung enthält bei der traditionellen Werkzeugarchitektur eine vollständige Kopie des Dokuments. Bei einer OMS-orientierten Werkzeugarchitektur könnte das Werkzeug theoretisch ganz ohne transiente Kopien auskommen: Es benötigt lediglich einen Verweis auf die Wurzel des aktuellen Dokuments. Die Dokumenteinträge, deren Attribute und ausgehenden Beziehungen können direkt aus der Objektbank ermittelt werden.

Praktisch müssen Daten aus der Objektbank jedoch im Werkzeug zwischengespeichert werden – aus zwei Gründen: Zum einen würde jede Zeichenoperation zumeist mehrere Zugriffe auf die Objektbank erfordern, was selbst bei einer hochperformanten Datenbank zuviel Zeit beanspruchen würde. Zum anderen wird die Darstellung eines Dokuments in einer graphischen Benutzungsschnittstelle aus zahlreichen Objekten zusammengesetzt, die vom verwendeten GUI-Framework vorgegeben werden. Das Werkzeug muß also eine Struktur von GUI-Objekten beim Start aufbauen und später aktualisieren, wenn die Daten in der Objektbank geändert wurden.

In *ToolFrame* [69] enthalten diese GUI-Objekte nur *Verweise* auf Objekte und Links in der Objektbank: Die Dokumentstruktur wird also zum Teil in einer transienten Kopie nachgebildet, lediglich die Eigenschaften von Einträgen und Beziehungen werden bei jeder Zeichenoperation aus der Objektbank ausgelesen und müssen folglich nicht im Werkzeug gepuffert werden (Abbildung 3.2). Dieser Ansatz ist allerdings nur praktikabel, wenn mit hoher Geschwindigkeit auf die Objektbank zugegriffen werden kann (lokal geladene Segmente, vgl. Abschnitt 2.5.5) und die Dokumente nicht mehr als einige Hundert Einträge besitzen.

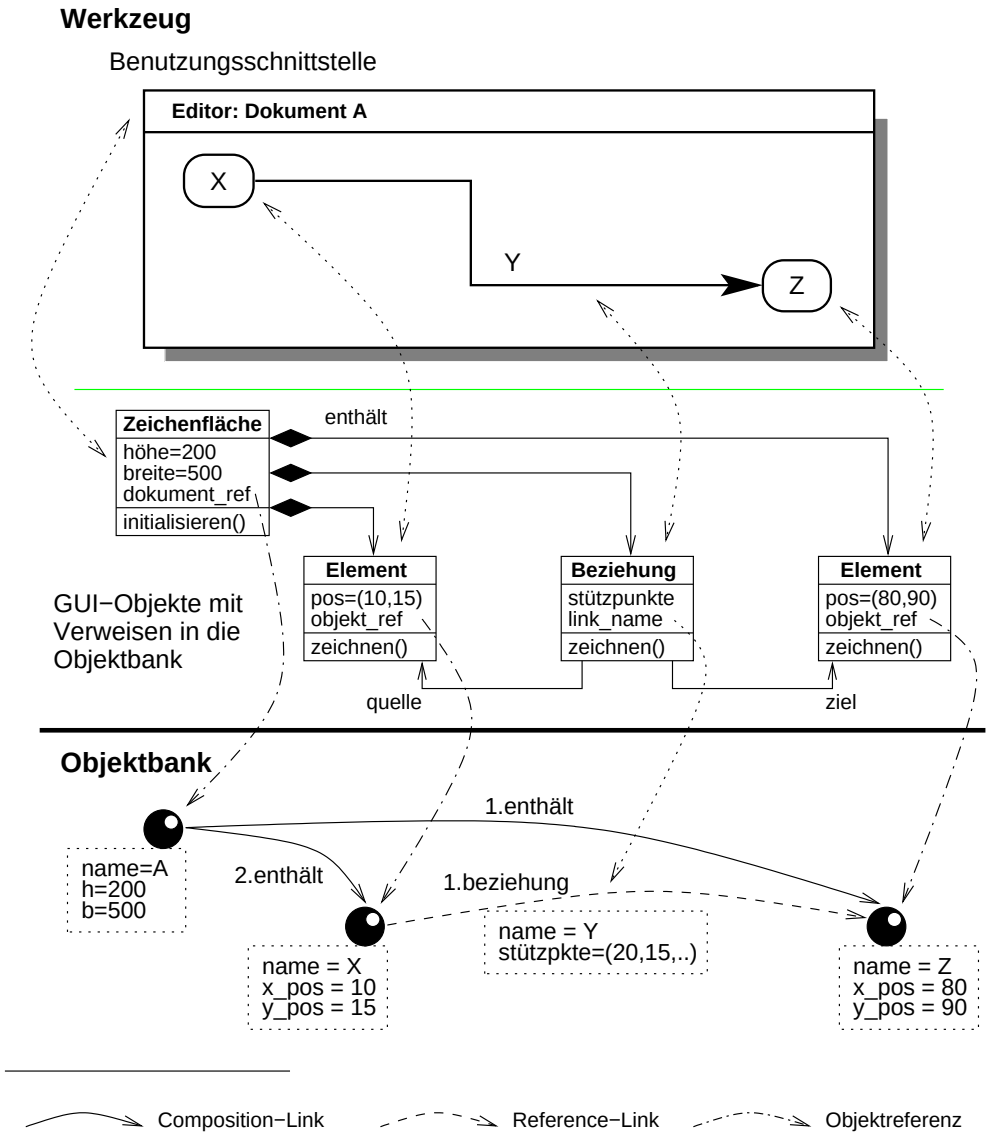


Abbildung 3.2: Dokumentverwaltung in *ToolFrame*

3.1.2 Grundlegendes Datenmodell

Im folgenden werden Werkzeuge betrachtet, die eine OMS-orientierte Architektur aufweisen und H-PCTE zur Datenverwaltung nutzen: Software-Dokumente werden im PCTE-Datenmodell als komplexe Objekte modelliert. Der Typ des *Wurzelobjekts* definiert die Attribute des Dokuments und die Komponentenbeziehungen (Composition-Links) zu den enthaltenen Einträgen.

Einträge werden durch Objekttypen mit Attributen modelliert und können ebenfalls Komponenten besitzen. Beziehungen zwischen Einträgen werden durch Reference-Links modelliert. Die Beziehungen können ebenfalls Attribute besitzen. Die Schlüsselattribute haben keine Semantik im Datenmodell der Dokumente: Sie dienen lediglich zur eindeutigen Benennung der Links, tragen aber keine dokumentspezifischen Informationen.

Abbildung 3.3 zeigt die graphische Darstellung eines SDS für Klassendiagramme. In Ab-

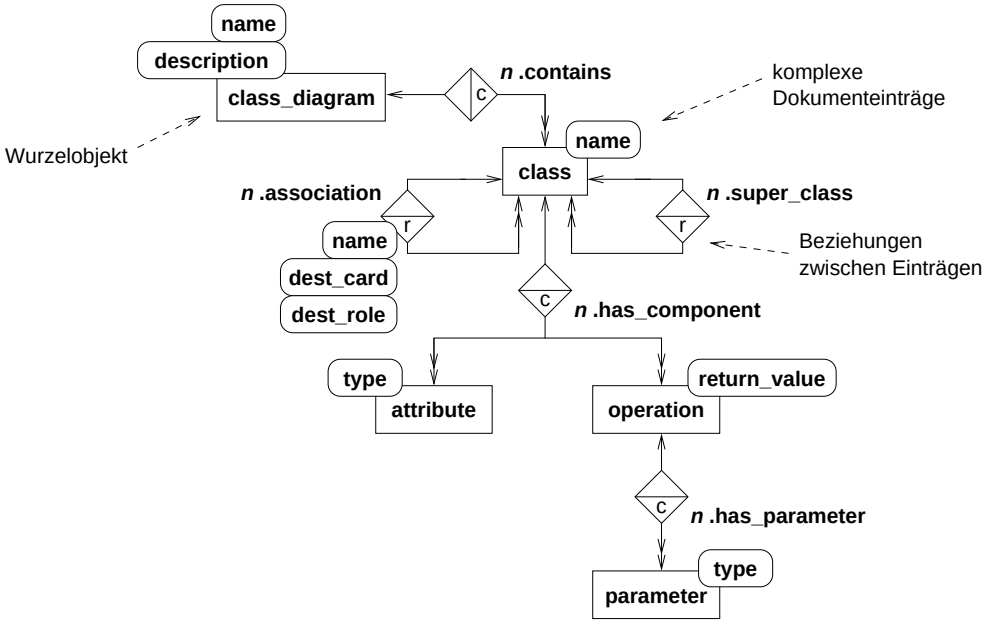


Abbildung 3.3: Vereinfachtes SDS für Klassendiagramme

bildung 3.4 ist dargestellt, wie ein Klassendiagramm in der Objektbank gespeichert wird. Außerdem sind die Werkzeuge skizziert, mit denen die Objektstruktur bearbeitet werden kann – mehr dazu im nächsten Abschnitt.

3.1.3 Schema-Interpretation

Um Dokumente feingranular in der Objektbank verwalten zu können, muß das Datenmodell der Dokumente im Datenbankschema beschrieben werden. Diese Beschreibung kann sehr umfangreich sein, wenn das Datenmodell eine große Zahl von Eintrags- und Beziehungstypen mit jeweils vielen Attributen enthält. In den Werkzeuge, mit denen die Dokumente bearbeitet werden, muß zwar bei einer OMS-orientierten Architektur die transiente Dokumentverwaltung nicht implementiert werden. Das Werkzeug muß aber eine Benutzungsschnittstelle zur Darstellung und zur Manipulation von Dokumenten anbieten, die stark vom Datenmodell abhängt – Schemainformationen sind also redundant sowohl im Datenbankschema als auch im Werkzeug vorhanden. Dies bereitet drei Probleme:

1. Die Implementierung der Benutzungsschnittstelle ist aufwendig und fehleranfällig; Inkonsistenzen zwischen Benutzungsschnittstelle und Datenbankschema führen zu Fehlern, die erst zur Laufzeit des Werkzeugs erkannt werden können.
2. Bei einer Änderung des Datenbankschemas muß auch die Benutzungsschnittstelle angepaßt werden, die Wartung erstreckt sich also auf zwei Ressourcen, deren Konsistenz gewahrt bleiben muß.
3. Während externe Sichten auf die Objektbank leicht definiert werden können, ist die Realisierung zugehöriger Benutzungsschnittstellen schwierig und führt zu komplizierten Verer-

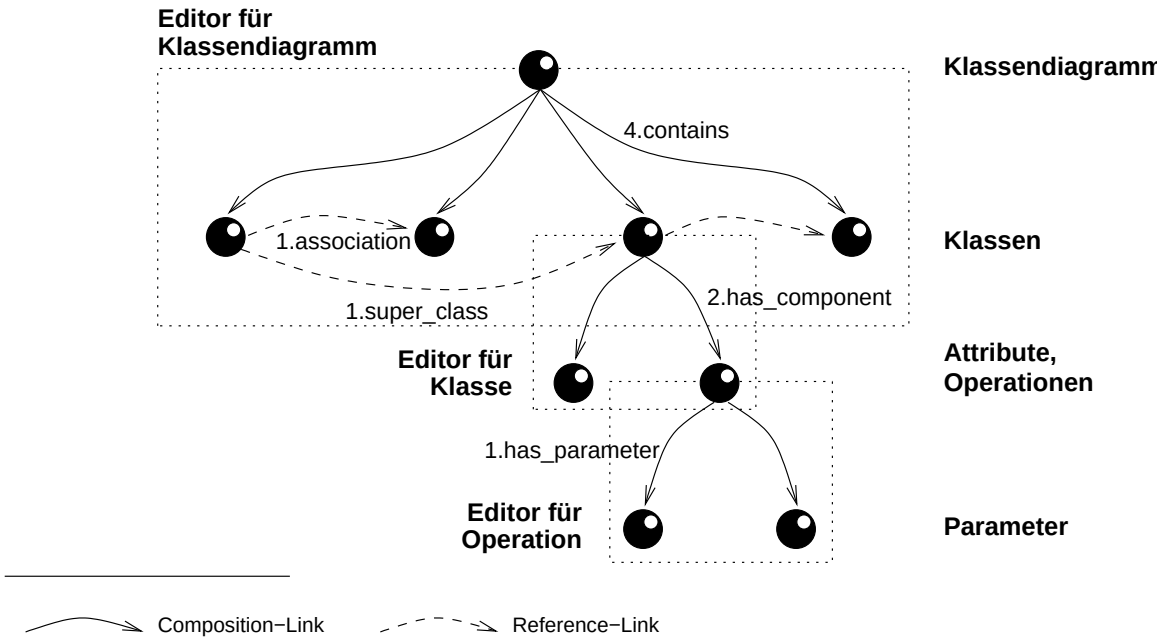


Abbildung 3.4: Klassendiagramm in der Objektbank und zugehörige Werkzeuge

bungshierarchien bei den Werkzeugkomponenten oder einer großen Zahl von Bausteinen, die passend zur Sicht zusammengesetzt werden müssen.

Die Probleme können gelöst werden, wenn auf die redundante Beschreibung des Datenmodells im Werkzeug verzichtet wird. Statt dessen *interpretiert* das Werkzeug das Datenbankschema – die Eigenschaften des Werkzeugs werden also durch das Datenbankschema bestimmt.

Generische Browser (z.B. [151, 301]) funktionieren mit allen Datenmodellen. Sie bieten generische listen-, baum- oder graphartige Darstellungen zur Anzeige von Daten und generische Funktionen zu deren Manipulation an. Dabei können (und dürfen) die Browser keine Annahmen über die Semantik der Daten machen: Alle Objekte und Beziehungen werden gleich behandelt. In *SurfBorD* [116] werden relationale Datenbankschemata ausgewertet und aus den Informationen passende Benutzungsschnittstellen zur Bearbeitung der Datenbankinhalte erzeugt. Eine Erweiterung um spezielle GUI etwa für Multi-Media-Daten ist vorgesehen. Die benötigten GUI-Komponenten werden von einem Administrator in der Datenbank abgelegt und zur Anzeige in *SurfBorD* geladen. Ähnlich funktionieren auch *aktive Dokumente* und *displets* [38]: Der Code für die Darstellung wird hier den Elementen einer XML-Datei zugeordnet und vom Browser geladen. Generatoransätze erzeugen Dialoge direkt aus dem Daten- oder Objektmodell einer Anwendung [18], so daß Modell und GUI automatisch konsistent sind. Im von Gruhn beschriebenen Ansatz [139] können die generierten Dialoge durch manuell erzeugte oder angepaßte Dialoge ergänzt werden.

In CASE-Werkzeugen müssen natürlich verschiedene Typen von Objekten und Beziehungen unterschiedlich dargestellt und auf unterschiedliche Weise manipuliert werden. Ein *generisches Werkzeug*, das sich dem Benutzer wie ein 'normales' CASE-Werkzeug präsentiert, muß also diese speziellen Eigenschaften der Typen ermitteln können:

– Werkzeugeigenschaften können direkt aus den Eigenschaften der Typen abgeleitet wer-

den. Welche Eigenschaften zu welchem Werkzeugverhalten führen, ist also im Werkzeug festgelegt: Besitzt ein Objekttyp ein Attribut name, könnte der Wert dieses Attributs zur Darstellung von Objekten in einer Liste verwendet werden; besitzt er die Attribute x_position und y_position, können Objekte des Typs in einer graphischen Darstellung verwendet werden.

– Zum anderen können die Werkzeugeigenschaften in separaten Beschreibungen festgelegt werden, die den Typen zugeordnet sind. Die Beschreibungen können in eigenen Ressourcen, in der Meta-Datenbank selbst oder im Werkzeugquelltext enthalten sein. Die letzte Möglichkeit nutzt *ToolFrame* [245]: Beschreibungen für Darstellungen und Werkzeuge werden beim Start von *ToolFrame* initialisiert und in einer Datenstruktur abgelegt. Für den Zugriff wird der Typname als Schlüssel verwendet.

Sichtenintegration

Die Produkte des Software-Entwicklungsprozesses können aus verschiedenen Blickwinkeln betrachtet werden, abhängig von der Rolle und Aufgabe des Betrachters und seinem Informationsbedürfnis. Diese verschiedenen Blickwinkel bereiten Probleme, wenn sie nicht *konsistent* zueinander sind: Ein objektorientierter Entwurf kann graphisch durch ein Klassendiagramm oder textuell in einer Baumdarstellung angezeigt und manipuliert werden. Wird im Klassendiagramm eine neue Klasse hinzugefügt, ein Klassenname geändert oder eine Beziehung gelöscht, müssen sich diese Änderungen auch auf die Baumdarstellung auswirken.

In *IPSEN* [212] würden im beschriebenen Szenario drei Dokumente verwendet: Ein *logisches Dokument*, das das objektorientierte Modell enthält und zwei *Darstellungsdokumente*, die Informationen für die textuelle bzw. graphische Darstellung des Modells enthalten. Eine Komponente in IPSEN ist dafür zuständig, logische und Darstellungsdokumente konsistent zu halten. Sachweh und Schäfer [287] schlagen Beziehungen zwischen Einträgen in feingranular modellierten Dokumenten vor, über die die Konsistenz sowohl innerhalb eines Dokuments als auch zwischen verschiedenen Dokumenten gesichert werden kann.

Beide Ansätze haben einen erhöhten Aufwand bei der Realisierung von Software-Entwicklungswerkzeugen zur Folge [68]. Dieser läßt sich durch die Nutzung des Sichtenmechanismus des OMS vermeiden: Statt Daten redundant zu speichern und für die Aktualisierung zu sorgen, werden Daten nur einmal gespeichert und verschiedene Sichten auf die Daten definiert. Die Sichten sind integriert, da alle Sichten auf einem gemeinsamen konzeptuellen Schema arbeiten (*Multiple View Integration*) [68].

In *ToolFrame* werden Sichten genutzt

– als Filter zum Ausblenden von Einträgen und Beziehungen im Dokument.

– zur Integration verschiedener Dokumente mit überlappenden Daten, etwa eines ER-Diagramms und dem zugehörigen Datenlexikon.

– zum Umschalten verschiedener Notationen eines Dokuments, etwa für ER-Diagramme. In den verschiedenen Sichten werden gemeinsame Typen importiert und umbenannt. Über den Typnamen werden Einträgen und Beziehungen die Symbole für die graphische Darstellung zugeordnet.

Generische Funktionen

In Abbildung 3.4 sind die Werkzeuge als gestrichelte Rechtecke angedeutet, mit denen die Objektstruktur editiert werden kann. Jedem Werkzeug ist ein Wurzelobjekt eines komplexen Objekts zugeordnet. Die Editierkommandos hängen von der Art des Werkzeugs ab. Die Objektstruktur kann wie folgt manipuliert werden:

- *Editieren der Attribute eines Objekts oder Links*: Es muß für jedes Attribut ein Eingabefeld vorhanden sein, mit dem der Wert des Attributs manipuliert werden kann. Dabei muß der Wertebereich des Attributs beachtet (Text, Zahl, Aufzählung) und ein passendes GUI-Element verwendet werden.
- *Editieren der Komponenten eines Objekts*: Es muß je ein Kommando zum Erzeugen von Instanzen aller Zielobjekttypen aller ausgehenden Composition-Links vorhanden sein, sowie ein Kommando zum Löschen von Komponenten.
- *Editieren der ausgehenden Beziehungen eines Objekts*: Es muß ein Kommando zum Erzeugen eines Reference-Links zu einem anderen Objekt vorhanden sein. Die Menge der Linktypen, von denen eine Instanz erzeugt werden kann, hängt von der Kombination der Typen von Ausgangs- und Zielobjekt ab. Zusätzlich muß ein Kommando zum Löschen von Beziehungen existieren.

Die Kommandos zur Manipulation der Objektbank können generisch implementiert werden und das Datenbankschema als Parameter betrachten. Somit ist kein Aufwand zur Anpassung der Kommandos an das jeweilige Datenbankschema nötig.

Die Werkzeugkommandos können in verschiedenen Benutzungsschnittstellen verwendet werden: Die Attribute eines Objekts können mit einem Formular bearbeitet werden; eine Tabelle enthält die Attribute mehrerer Objekte und erlaubt es, sowohl die Attributwerte zu editieren, als auch neue Einträge zu erzeugen oder Einträge zu löschen. Die Menge der ausgehenden Beziehungen eines Objekts kann als Liste der Zielobjekte dargestellt werden oder in einem graphischen Editor werden Beziehungen als Linien zwischen den graphisch dargestellten Dokumenteinträgen angezeigt und manipuliert. Mit einer Baumdarstellung können mehrere Ebenen eines komplexen Objekts editiert werden (Klassen, Operationen, Parameter). Dies ist auch in einer graphischen Darstellung möglich, wenn Attribute und Operationen innerhalb eines Klassensymbols manipuliert werden können.

In *ToolFrame* gibt es zwei Arten von Editoren: In graphischen Editoren werden Dokumenteinträge auf einer Zeichenfläche angeordnet; in einem Gitter-Editor werden sie als Piktogramme in einem Raster dargestellt. Graphische Editoren sind für verschiedene Diagrammtypen, wie ER- und Klassendiagramme, verfügbar; Gitter-Editoren werden zur Verwaltung von Projekten und Dokumenten genutzt. Mit den Werkzeugen kann jeweils nur eine Ebene in der Komponentenhierarchie bearbeitet werden (vgl. Abbildung 3.4): Im Editor für Klassendiagramme können nur Klassen und ihre Beziehungen manipuliert werden; um die Attribute und Operationen einer Klasse zu bearbeiten, muß ein separater Editor (mit der Klasse als Wurzelobjekt) geöffnet werden.

Konsistenzbedingungen

Durch das Datenbankschema wird die Struktur von Dokumenten beschrieben. Dokumente, die im OMS verwaltet werden, sind somit zwangsläufig konsistent mit dem Datenbankschema. Damit ist sichergestellt, daß ein Dokument nur Einträge und Beziehungen der erlaubten

Typen enthält, und (bei Linktypen mit referentieller Integrität) daß jeder Link ein Zielobjekt besitzt und daß Schlüsselattribute eindeutige Werte haben.

Mit Kardinalitäten kann die Zahl der in Beziehung stehenden Objekte eingeschränkt werden: Linktypen ohne Schlüsselattribut können null- oder einmal vorkommen; Linktypen mit Schlüsselattribut auch mehrfach. Die minimale und maximale Anzahl ausgehender Links eines Typs kann pro Objekttyp spezifiziert werden. Die Obergrenze wird beim Erzeugen von Links geprüft, die Untergrenze beim Löschen. Letztere Prüfung hat zur Folge, daß ein Link nicht mehr gelöscht werden kann, wenn damit die untere Grenze unterschritten würde. Diese Einschränkung kann umgangen werden, wenn das Werkzeug ein spezielles Kommando verwendet, das den Link durch einen anderen ersetzt, statt ihn zu löschen.

Im Datenbankschema kann also nur eine Teilmenge der Konsistenzbedingungen eines Dokumenttyps definiert werden. *Analysatoren* prüfen weitere und komplexere Konsistenzbedingungen – und zwar automatisch in Folge einer Änderung oder auf Anforderung durch den Benutzer. In *ToolFrame* sind Analysatoren als separate Werkzeuge realisiert [68]. Alternativ könnte eine Analysefunktion in einen Editor integriert werden, somit die Prüfung ohne Wechsel des Werkzeugs aufgerufen und das Ergebnis direkt im Editor angezeigt werden.

3.1.4 Anforderungen an den ToolFrame-Nachfolger

Ein Ansatz zur Konstruktion von CASE-Werkzeugen, der auf H-PCTE basiert und in *ToolFrame* entwickelte Ideen wie die OMS-orientierte Werkzeugarchitektur und die Schema-Interpretation nutzt, kann als *ToolFrame*-Nachfolger betrachtet werden. Folgende Anforderungen an einen solchen Nachfolger haben die Ziele und Entwurfsentscheidungen der Arbeit beeinflußt:

1. *Trennung zwischen Framework und Umgebung*: *ToolFrame* ist gleichzeitig ein Framework zum Bau von CASE-Werkzeugen und die einzige Instanz dieses Frameworks. Ziel ist es, beides zu trennen, so daß mehrere Instanzen des Frameworks parallel existieren und unabhängig erweitert werden können, andererseits Änderungen und Erweiterungen am Framework für alle Instanzen verfügbar sind.
2. *Mehrbenutzerfähigkeit, Verteilung*: CASE-Werkzeuge sollen mehrbenutzerfähig sein, also die parallele Arbeit mehrerer Benutzer auf überlappenden Daten ermöglichen. Um dies technisch zu realisieren, sollen geeignete Mechanismen von H-PCTE ausgenutzt werden; hierzu gehören Zugriffsrechte und Transaktionen in Verbindung mit feingranularen Sperren. Damit die Benutzer von unterschiedlichen Standorten aus auf die Objektbank zugreifen können, müssen zumindest die Benutzungsschnittstellen der Werkzeuge räumlich verteilt werden können. Werkzeuge sollten also über Internet-Protokolle auf die Objektbank zugreifen können und den verteilten Benachrichtigungsmechanismus von H-PCTE nutzen.
3. *Werkzeugkonstruktion*: Um den Bau und die Anpassung von Werkzeugen zu erleichtern, ergeben sich folgende Anforderungen:
   - Die minimale Größe der Werkzeugbausteine muß verringert werden: Statt ganzer Werkzeuge müssen Werkzeugteile, wie Darstellungskomponenten und Kommandos, einzeln ausgetauscht werden können.

– Die Zusammensetzung von Werkzeugen aus den Bausteinen soll erleichtert werden und (zum großen Teil) ohne Programmierung möglich sein. Auch soll der Bau von Werkzeugen durch Meta-Werkzeuge unterstützt und erleichtert werden.
– Die Konsistenz zwischen Werkzeug und Datenbankschema ist auch in einer Interpreter-Architektur ein Problem, sobald Informationen aus dem Schema direkt im Werkzeug verwendet werden. Dies geschieht etwa bei der Festlegung des Attributs, durch das Objekte eines Typs in einer Liste dargestellt werden, oder bei der Festlegung der Attribute, die in den Spalten einer Tabelle erscheinen. Beim Werkzeugbau sollen Konsistenzverletzungen einfach zu vermeiden oder durch Prüfwerkzeuge feststellbar sein.

3.1.5 Einige Beispiele für Werkzeuge

Nach den eher abstrakten Anforderungen des vorigen Abschnitts werden nun die Anforderungen an den Werkzeugbau anhand konkreter Werkzeuge erläutert und motiviert. Die Werkzeuge stammen aus *PI-SET*, einer *Upper-CASE*-Werkbank, die hauptsächlich als Experimentierumgebung zur Validierung des vorgestellten Ansatzes dient – sowohl im Hinblick auf seine Flexibilität und Praktikabilität, als auch auf die Verwendbarkeit der resultierenden Werkzeuge. *PI-SET* wurde auch im Rahmen der Softwaretechnik-Ausbildung in einer Lehrveranstaltung eingesetzt [248]. Es kann (und soll) aber nicht als vollwertige und stabile CASE-Umgebung betrachtet werden.

Datenmodellierung mit ER-Diagrammen

Die Datenmodellierung ist eine zentrale Aufgabe beim Entwurf von Informationssystemen. ER-Diagramme schaffen einen Überblick über die Daten, mit denen das zu entwerfende System arbeiten soll, und ermöglichen einen leichten Übergang zur Realisierung des Systems auf Basis eines relationalen Datenbanksystems. Beides wird von *PI-SET* unterstützt – mit textuellen und graphischen Editoren sowie einem Generator für Tabellendefinitionen.

Werkzeugschemata. Die Eigenschaften eines Werkzeugs werden durch eine Werkzeugspezifikation festgelegt. Um eine Werkzeugsammlung zu spezifizieren, müssen also mehrere Werkzeugspezifikationen erzeugt werden. Dabei können Teile dieser Spezifikationen in mehreren Werkzeugen verwendet werden, um gemeinsame Eigenschaften zu beschreiben. Werkzeugspezifikationen werden im folgenden *Werkzeugschemata* genannt. In Abbildung 3.5 sind die Werkzeugschemata dargestellt, mit denen die ER-Werkzeuge spezifiziert werden. Werkzeugschemata werden mit Paketsymbolen aus der UML dargestellt. Zwei Arten von Beziehungen können zwischen Werkzeugschemata verlaufen: Ein gestrichelter Pfeil mit leerer Spitze symbolisiert, daß das Werkzeugschema, von dem der Pfeil ausgeht, Definitionen aus dem Ziel-Werkzeugschema importiert – es also nutzt und erweitert, ähnlich einer Vererbung zwischen Klassen.

Ein durchgezogener Pfeil symbolisiert eine Aufruf- oder Navigationsbeziehung, die auch als Nutzt-Beziehung auf Instanzebene betrachtet werden kann: Im Werkzeug erscheinen Kommandos zum Aufruf von Werkzeugen aus dem Zielschema. Dieser Aufruf kann auch eine Navigation zu den Dokumenten beinhalten, die im Zielschema beschrieben werden.

Die Werkzeugschemata aus Abbildung 3.5 sind:

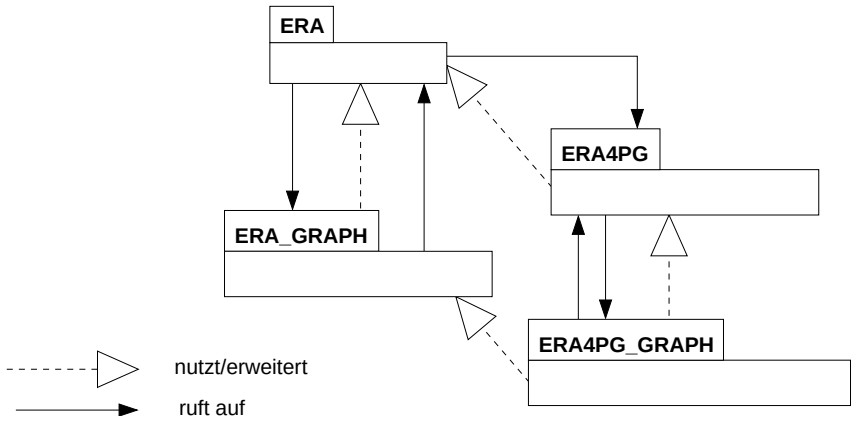


Abbildung 3.5: Werkzeugschemata für ER-Werkzeuge

– ERA spezifiziert das Datenmodell der ER-Diagramme und enthält Werkzeugparameter, die allgemeine Eigenschaften wie die Namensattribute von Einträgen oder das Format von Eingabefeldern festlegen. Auf dem Datenmodell arbeiten generische textuelle Werkzeuge (Abbildung 3.6, das Werkzeugschema wird jeweils links oben im Fenster angezeigt, daneben der Typ des Dokuments – hier 'ERD'). Die textuelle Sicht kann auch als Datenlexikon aufgefaßt werden – Datenlexikon und Diagramm sind dabei automatisch konsistent, da sie unterschiedliche Sichten auf einen gemeinsamen Datenbestand darstellen.

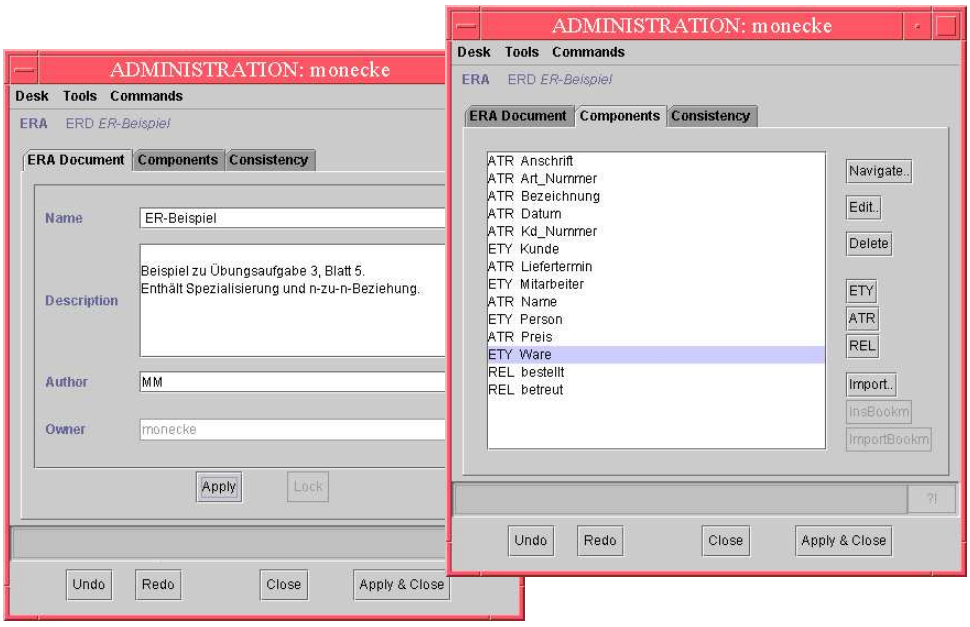


Abbildung 3.6: Textuelles Werkzeug für ein ER-Diagramm

– ERA_GRAPH spezifiziert den graphischen ER-Editor (Abbildung 3.7). Das Werkzeugschema enthält also Werkzeugparameter, die die Komponenten für die graphische Darstellung festlegen, sowie ein Kommando zum Export von Grafik-Dateien im 'Commands'-Menü des Werkzeugs.

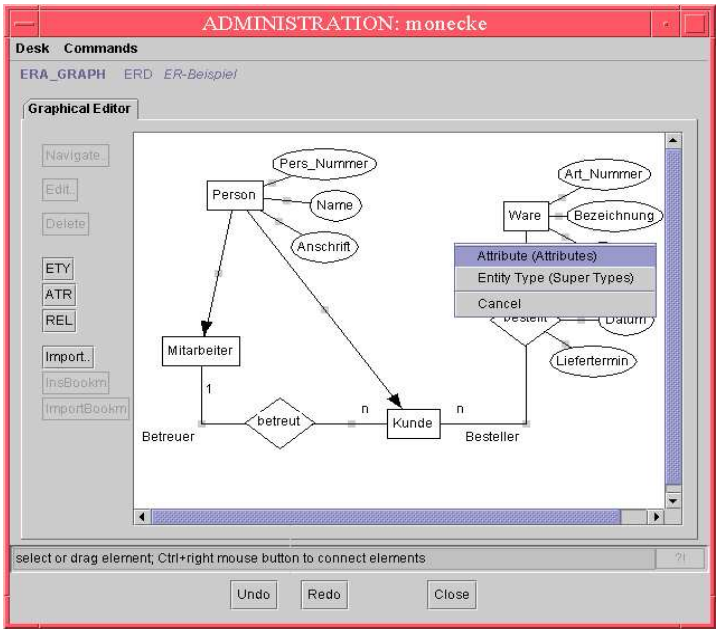


Abbildung 3.7: Graphischer ER-Editor

- ERA4PG spezifiziert Werkzeuge, mit denen ein ER-Diagramm so erweitert werden kann, daß direkt Datenbanktabellen aus dem Diagramm erzeugt werden können – hier für das DBMS *Postgres* [276]. ERA4PG erweitert dazu das ERA-Werkzeugschema um zusätzliche Datentypen und ordnet den ER-Diagrammen ein Kommando zum Generieren der Tabellendefinitionen zu. Attribute werden in einer Tabelle dargestellt (Abbildung 3.8). In der Tabelle können Eigenschaften der Attribute editiert werden, die für das Generieren der Tabellendefinitionen relevant sind, nämlich die Schlüsseleigenschaft, die Möglichkeit von Nullwerten und Einschränkungen des Wertebereichs.
- ERA4PG_GRAPH beschreibt einen graphischen Editor, in dessen Darstellung auch Informationen aus dem ERA4PG-Werkzeugschema angezeigt werden – etwa die Kennzeichnung von Schlüsselattributen durch Unterstreichen. Er importiert dazu Definitionen sowohl aus dem textuellen ERA4PG-Schema als auch aus dem graphischen ERA_GRAPH-Schema.

Konsistenztests. Mit Konsistenztests kann die Korrektheit und Qualität von Dokumenten überprüft werden. Das Ergebnis eines Konsistenztests kann sowohl in einer graphischen Darstellung (Markierung fehlerhafter Einträge) als auch in einer textuellen Darstellung (Anzeige von Fehlermeldungen und Hinweisen) angezeigt werden. Im Fenster in Abbildung 3.6 befindet sich die textuelle Darstellung auf der Tabulatorkarte *Consistency*. Sinnvolle Prüfungen sind hier:

- Alle Einträge müssen sinnvoll benannt sein – also etwa mit einem Buchstaben beginnen und mindestens drei Zeichen lang sein.
- Generalisierungsbeziehungen dürfen keine Zyklen enthalten.
- Ein Beziehungstyp muß mit mindestens zwei Entitätstypen verbunden sein.
- Ein Attribut muß mit genau einem Entitäts- oder Beziehungstyp verbunden sein.

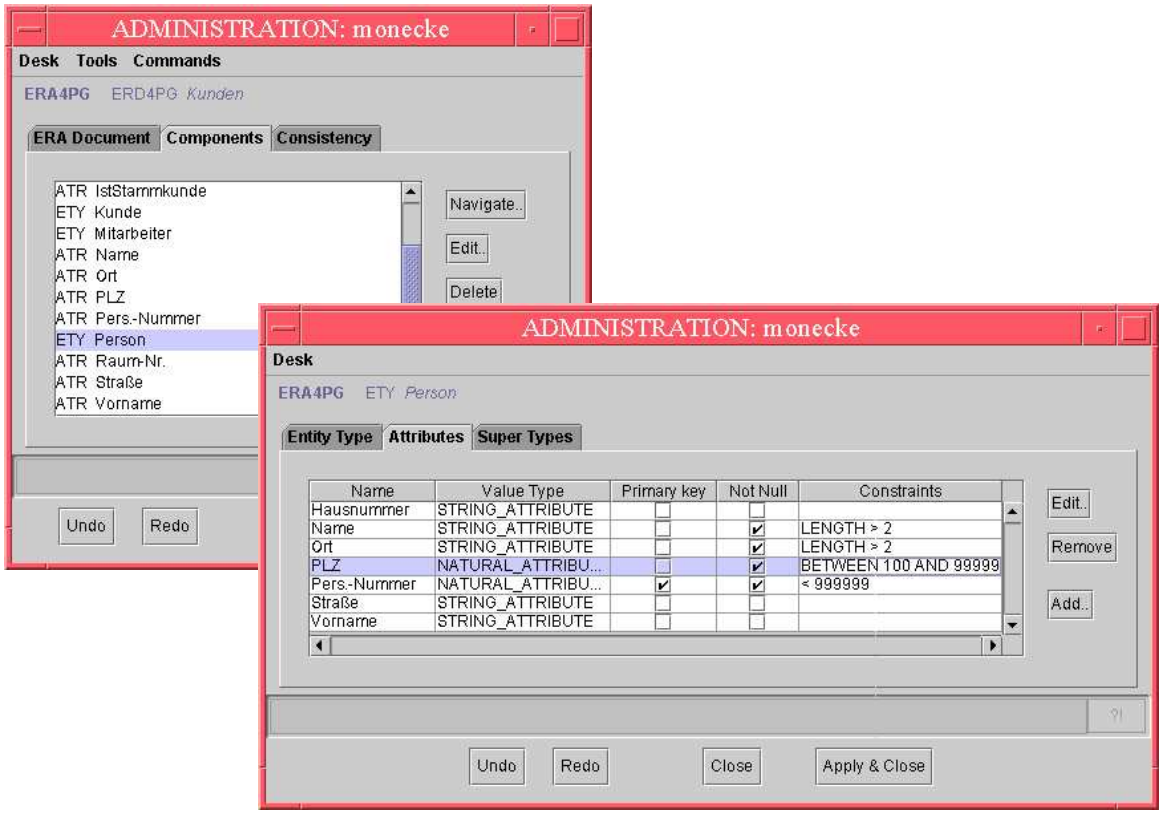


Abbildung 3.8: Tabellenwerkzeug zur Bearbeitung von Attributen

– Entitätstypen müssen mindestens ein Attribut besitzen oder in eine Beziehung involviert sein.

Im ERA4PG-Schema müssen die Konsistenzbedingungen erweitert werden: Beziehungstypen müssen hier Kardinalitäten besitzen, Attribute einen gültigen Datentyp; *constraints* müssen korrekt formuliert sein.

Werkzeugkomponenten. Werkzeuge sind modular aus Komponenten aufgebaut. Zu unterscheiden sind generische Komponenten, die in vielen Werkzeugen eingesetzt werden können, und werkzeugspezifische Komponenten, die Funktionen oder Darstellungen eines bestimmten Werkzeugs implementieren. Im Beispiel sind dies:

– Komponenten, die die graphische Darstellung der Dokumenteinträge implementieren – dies sind Attribute, Entitäts- und Beziehungstypen.
– Komponenten, die die graphische Darstellung von Beziehungen zwischen Einträgen implementieren – hier die Generalisierungsbeziehung sowie die Beziehungen zwischen dem ER-Beziehungstyp und den Entitätstypen. Letztere besitzen Eingabefelder, mit denen Kardinalität und Rolle angezeigt und in der graphischen Darstellung geändert werden können.
– Eine Komponente, die das Kommando zum Exportieren von Tabellendefinitionen implementiert. Im Kommando wird das ER-Modell traversiert und es werden für alle Entitäts- und Beziehungstypen SQL-Anweisungen zum Erzeugen der Tabellen ausgegeben.

– Komponenten, die die Konsistenztests implementieren. Für jeden Test muß eine Komponente implementiert oder eine generische Komponenten passend parametrisiert werden. Beispiele für generische Tests sind die Prüfung von Attributwerten oder ausgehenden Links. Die Komponenten werden in ein einfaches generisches Prüfwerkzeug geladen, die Prüfungen ausgeführt und das Ergebnis angezeigt. Die Komponenten können direkt über API-Operationen auf die Objektbank zugreifen oder Anfragesprachen [166] nutzen.

Analyse, Entwurf und Implementierung mit UML

Die *UML* [265] enthält zahlreiche Dokumenttypen, die in der Analyse und im Entwurf eingesetzt werden können. In *PI-SET* sind Werkzeuge für folgende Dokumenttypen enthalten:

– Paket- und Klassenstrukturen
– Klassendiagramme
– Objektdiagramme
– Zustandsübergangsdiagramme
– Anwendungsfall-Diagramme
– Kollaborationsdiagramme

Zwischen den Diagrammen verschiedenen Typs können Beziehungen existieren – etwa haben die Objekte im Objektdiagramm Verweise auf die zugehörige Klasse in der Klassenstruktur; im Kollaborations- oder Zustandsübergangsdiagramm werden Operationen der Klassen verwendet. In *PI-SET* werden diese Beziehungen auf Linktypen abgebildet, die zwischen den SDS der einzelnen Dokumenttypen verlaufen. Somit ist die Wahrung der Konsistenz zwischen den Dokumenttypen leicht möglich.

Im folgenden wird näher auf die Werkzeuge zur Bearbeitung von Paket- und Klassenstrukturen in textueller und graphischer Darstellung eingegangen.

Werkzeugschemata. Abbildung 3.9 zeigt die Werkzeugschemata für den textuellen Klassenstruktur- und den graphischen Klassendiagramm-Editor:

Unterschieden werden einerseits Werkzeuge für die verschiedenen Phasen Analyse, Entwurf und Programmierung (CSD_OOA, CSD_OOD bzw. CSD_OOP). Die Werkzeuge unterscheiden sich im Datenmodell, den Kommandos und den Konsistenztests:

– Im Entwurf werden mehr Informationen verwaltet als in der Analyse, hierzu zählen die Sichtbarkeit von Modellelementen, Typen von Attributen sowie die Signatur und der Rückgabewert von Operationen.
– Im Entwurf müssen mehr und schärfere Konsistenzbedingungen geprüft werden, um einen reibungslosen Übergang zur Implementierung zu ermöglichen.
– In der Implementierungsphase kann der Quelltext von Operationen bearbeitet werden. Genauer wird vom Werkzeug nur der Übergang von der Entwurfs- in die Implementierungsphase unterstützt, da aus der Klassenstruktur und den eingegebenen Operationsrümpfen ein Klassengerüst generiert werden kann. Umgekehrt kann der Quelltext vorhandener Klassen eingelesen und daraus eine Klassenstruktur im Repository erzeugt werden. In beiden Fällen wird in *PI-SET* die Programmiersprache *Java* unterstützt.

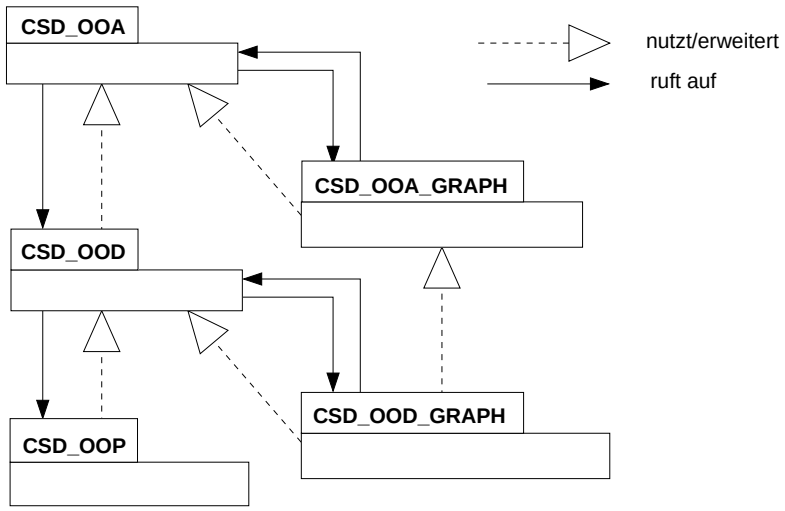


Abbildung 3.9: Werkzeugschemata für Klassenstruktur- und Klassendiagramm-Editoren

Zum anderen werden graphische und textuelle Werkzeuge unterschieden. Anders als bei den ER-Werkzeugen stellen die graphischen Werkzeuge (CSD_OOA_GRAPH, CSD_OOD_GRAPH) die Dokumente nicht nur anders dar, sondern verwenden auch ein anderes Datenmodell: Pakete, Klassen und Schnittstellen werden in der Pakethierarchie verwaltet; die Diagramme enthalten lediglich *Verweise* auf die Einträge in der Paketstruktur. Dadurch können Diagramme auf bestimmte Aspekte des Systems eingeschränkt und damit übersichtlicher gestaltet werden. Abbildung 3.10 zeigt einen vereinfachten Ausschnitt aus den zugehörigen SDS.

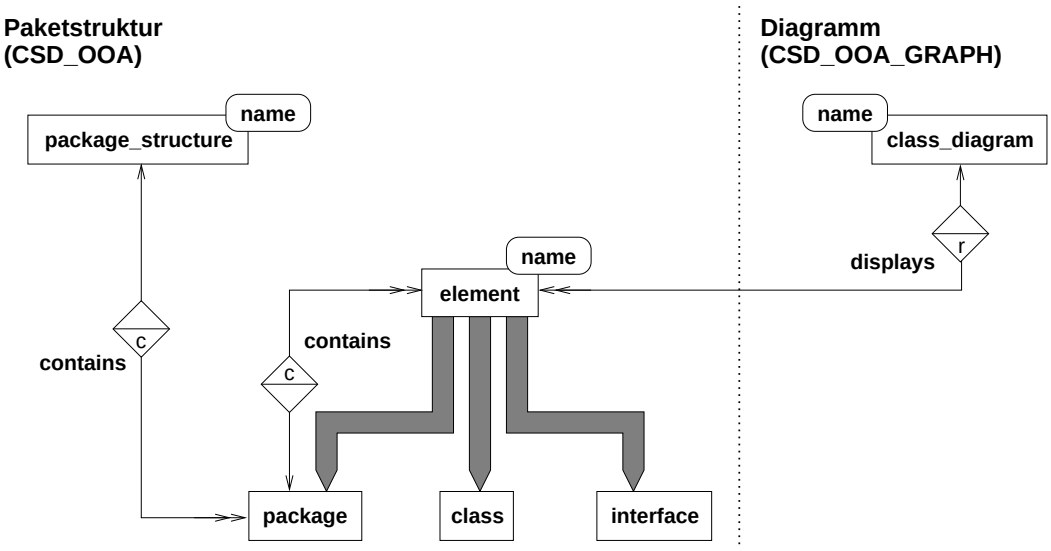
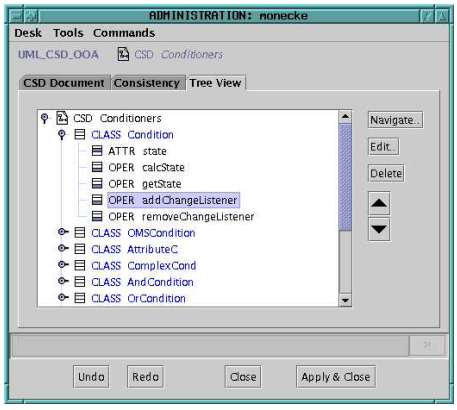


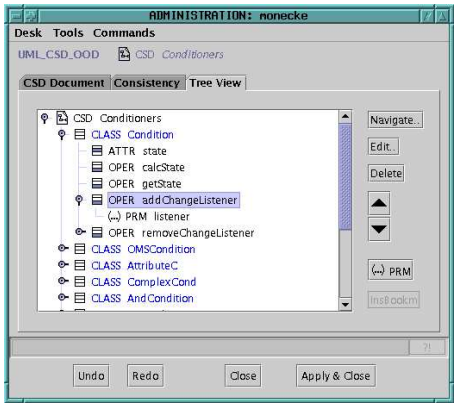
Abbildung 3.10: SDS für Pakethierarchie und Klassendiagramme

Der Linktyp displays vom Klassendiagramm zu den Elementen hat die Kategorie *reference*, ist also nur ein Verweis. Die Elemente werden in Paketen verwaltet, die wiederum Kompo-

nenten des Wurzelobjekttyps package_structure sind. Die contains-Links haben die Kategorie *composition*. In Abbildung 3.11 sind textuelle, in Abbildung 3.12 graphische Werkzeuge für Analyse und Entwurf dargestellt[1].



(a) Textuelles OOA-Werkzeug: Parameter sind nicht sichtbar und können nicht bearbeitet werden



(b) Textuelles OOD-Werkzeug: Parameter können editiert werden

Abbildung 3.11: Textuelle Werkzeuge für die Analyse und den Entwurf

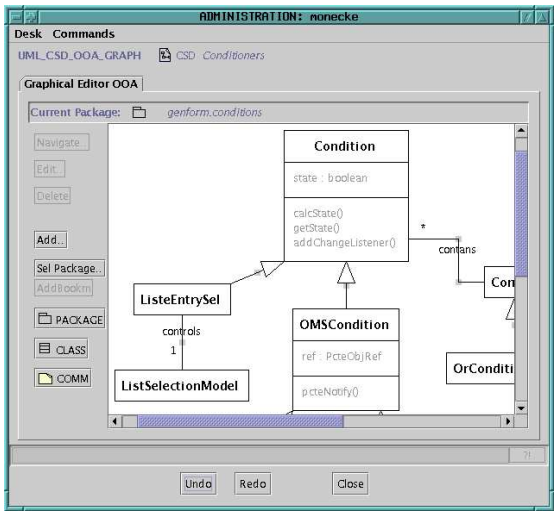
Somit unterscheidet sich der graphische Editor für Klassendiagramme von dem für ER-Diagramme: Im Klassendiagramm-Editor werden keine Komponenten des Dokuments angelegt, sondern nur Verweise auf vorhandene Objekte in der Paketstruktur. Allerdings kann ein aktuelles Paket ausgewählt werden, in dem auch Einträge erzeugt und gelöscht werden können – neue Einträge werden dabei automatisch in das Diagramm übernommen, also auch der Reference-Link vom Dokument zum Eintrag erzeugt. Die beschriebenen Funktionen sind dabei nicht dokumentspezifisch, sondern generisch und werden durch die Werkzeugschemata gesteuert.

Die gleichen Funktionen werden auch genutzt, um in einem Dokument Verweise auf Einträge eines anderen Dokuments verwalten zu können: In diesem Fall gehen von der Dokumentwurzel sowohl Composition-Links auf die eigenen Einträge aus als auch Reference-Links auf die Einträge des anderen Dokuments. Auf diese Weise werden etwa im graphischen SDS-Editor definierte und importierte Typen unterschieden.

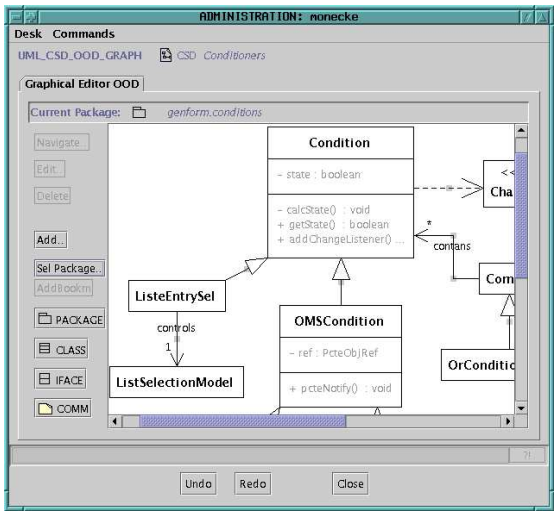
Konsistenztests. Sinnvolle Konsistenzbedingungen für objektorientierte Analyse- und Entwurfsdiagramme sind folgende:

– Alle Dokumenteinträge müssen einen sinnvollen und eindeutigen Namen besitzen. Bei Attributen und Operationen müssen die Namen pro Klasse oder Schnittstelle eindeutig sein sowie bei Attributen innerhalb der Vererbungshierarchie.

[1] Natürlich liegen die Unterschiede zwischen Analyse- und Entwurfsdiagrammen nicht nur in der Darstellung; aber durch die Integration der Analyse- und Entwurfssicht können Analysediagramme leicht als Basis für den Entwurf verwendet werden.



(a) Graphisches OOA-Werkzeug: Darstellung als Analysediagramm



(b) Graphisches OOD-Werkzeug: Darstellung als Entwurfsdiagramm mit Sichtbarkeiten, gerichteten Assoziationen, Attributtypen und Rückgabetypen von Operationen

Abbildung 3.12: Graphische Werkzeuge für die Analyse und den Entwurf

– Klassen müssen mindestens ein Attribut, eine Operation oder eine ausgehende Beziehung besitzen.
– Vererbungshierarchien sollten nicht zu tief sein. Eine Überschreitung der gewählten Grenze sollte aber eher als Warnung aufgefaßt werden.

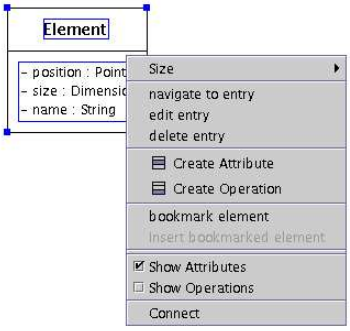
Die Bedingungen können abhängig von der Entwicklungsphase gelockert oder verschärft werden – sie werden daher in den verschiedenen Werkzeugschemata variieren. Es wurden auch zahlreiche Eigenschaften objektorientierter Entwürfe und Systeme identifiziert, mit Meßverfahren ermittelt [53] und ihre Auswirkungen auf Qualität [24], Wartbarkeit [225] und das Projekt-Management [297, 321] untersucht. Solche Messungen könnten ebenfalls vom Prüfwerkzeug ausgeführt und aus den Ergebnissen Hinweise für die Entwickler abgeleitet werden. Problematisch ist hier, daß die Ergebnisse der Konsistenz- und Qualitätsprüfungen nicht in der graphischen Darstellung veranschaulicht werden können. Um dies zu ermöglichen, müßten

– die Datenmodelle der Dokumente in einer speziellen Sicht so erweitert werden, daß die Meßergebnisse in der Objektbank gespeichert werden können.
– Komponenten implementiert werden, die die Messungen und Prüfungen durchführen und die Ergebnisse in der Objektbank speichern.
– Komponenten für die Darstellung der Dokumente implementiert werden, die die gemessenen Eigenschaften visualisieren.

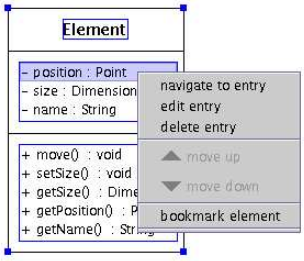
Eine Variante des Klassendiagramm-Editors würde dann mit dem erweiterten Datenmodell arbeiten, Prüfkommandos anbieten und die Ergebnisse graphisch und textuell darstellen.

Werkzeugkomponenten. Die Zahl der werkzeugspezifischen Komponenten ist für die Klassendiagramm-Editoren größer als für den ER-Editor, da es mehr Eintragstypen gibt und sich die Komponenten für die graphische Darstellung von Analyse- und Entwurfsdiagrammen unterscheiden. Ein Beispiel für ein komplexes graphisches Element ist das Klassensymbol (Abbildung 3.13).

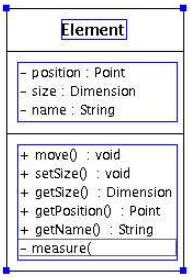
(a) Kompaktes Klassensymbol



(b) Klassensymbol mit eingeblendeten Attributen und Menü zum Bearbeiten der Klasse



(c) Klassensymbol mit Menü zum Bearbeiten der Attributliste



(d) Editieren einer Operation im Klassensymbol

Abbildung 3.13: OOD-Klassensymbol

Das Klassensymbol zeigt den Namen, den Stereotyp und die Liste von Attributen und Operationen einer Klasse an, wobei die Attribute und Operationen ein- und ausgeblendet werden können (Abbildung 3.13(b)). Die Breite des Symbols wird anhand der Länge des Klassennamens und der Attribut- und Operationsnamen bestimmt, wobei eine vorgegebene maximale Breite nicht überschritten wird. Die Höhe des Symbols hängt von der Anzahl der Attribute und Operationen ab; auch hier ist die maximale Höhe begrenzt. Alternativ kann die Größe des Elements manuell beeinflußt werden (Menüpunkt 'Size').

Im Symbol kann der Klassenname direkt editiert werden. Außerdem können Attribute und Operationen über das Menü in Abbildung 3.13(b) erzeugt werden. Kommandos zum Bearbeiten der Einträge in der Attribut- und Operationsliste stehen im Menü in Abbildung 3.13(c) zur Verfügung. Die Listeneinträge können auch direkt im Symbol editiert werden

(Abbildung 3.13(d)): Hier wird die Sichtbarkeit, der Name und der Rückgabetyp der Operation measure im Eingabefeld bearbeitet – ein Parser überprüft die Eingabe und setzt die Attributewerte in der Objektbank.

Zustandsmodellierung mit Petri-Netzen

PI-SET unterstützt die Zustandsmodellierung mit einfachen Stellen/Transitionsnetzen. Petri-Netze sind also zunächst ein weiterer Dokumenttyp – die Werkzeuge für Dokumente dieses Typs werden realisiert mit Werkzeugschemata und speziellen Werkzeugkomponenten (Abbildung 3.14). Eine Besonderheit der Petri-Netz-Dokumente ist, daß sie simuliert werden können – *PI-SET* enthält dazu einen (sehr einfachen) Simulator, der die Möglichkeiten zur flexiblen Erweiterung der Umgebung demonstrieren soll. Bei der Simulation kommen verschiedene *genform*-Konzepte zum Einsatz: Werkzeugsichten, dokumentspezifische Darstellungskomponenten und ein Simulationsagent.

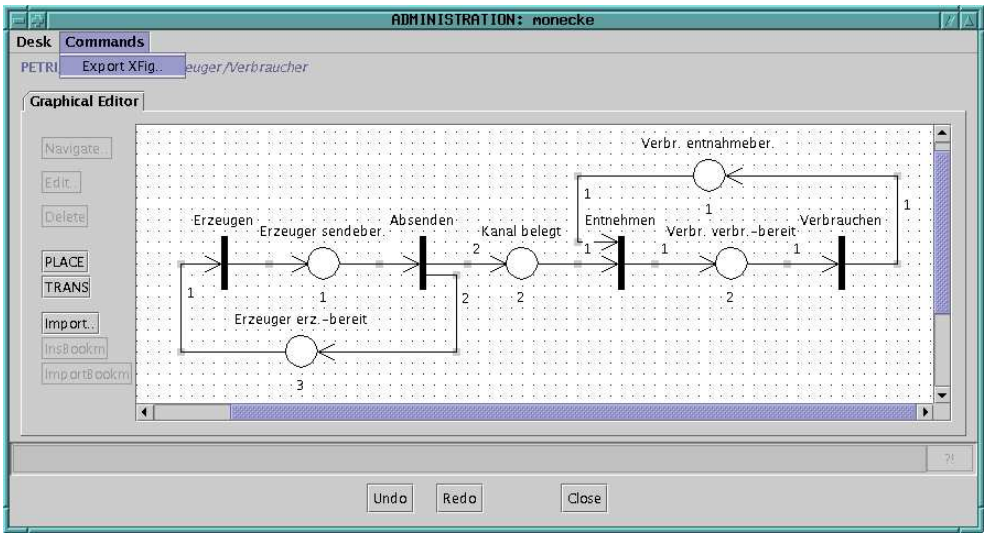


Abbildung 3.14: Editor für Petri-Netze

Werkzeugschemata. In Abbildung 3.15 sind die Werkzeugschemata dargestellt, mit denen Werkzeuge für Petri-Netze spezifiziert werden. Neben den Werkzeugschemata für textuelle (PETRI) und graphische Werkzeuge (PETRI_GRAPH) definiert das Werkzeugschema PETRI_SIM die Werkzeugsicht des Simulators: Er zeigt das Netz mit seinem aktuellen Zustand an und ermöglicht dessen Beeinflussung (Abbildung 3.16): Die Markierung der Plätze wird durch Zahlen dargestellt; der Zustand der Transitionen durch Farben. Das 'Commands'-Menü enthält Kommandos zur Steuerung der Simulation.

- Das Datenmodell ist erweitert um Typen, mit denen der aktuelle Zustand des Netzes verwaltet werden kann, also die Markierung der Plätze und der Zustand von Transitionen.
- Typrechte verhindern die Änderung des Netzes – somit sind im Werkzeug keine Editieroperationen außer denen zur Beeinflussung des Zustands verfügbar – erkennbar an der Werkzeugleiste am linken Rand. Da der Unterschied im Werkzeugschema liegt, kann die generische Editorkomponente ohne Änderung verwendet werden.

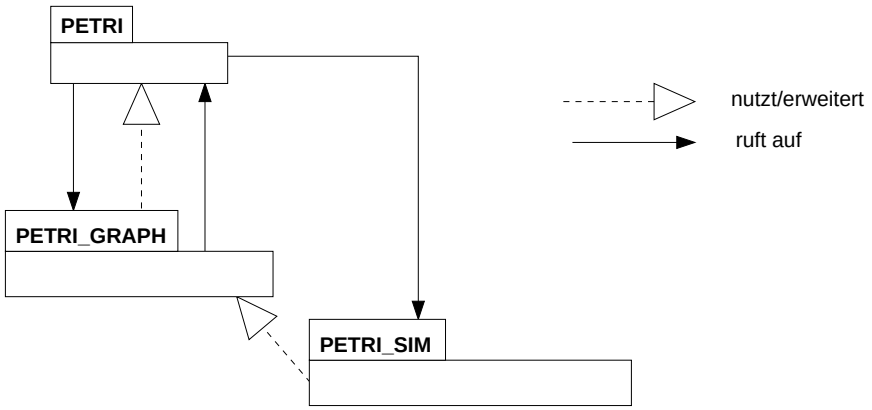


Abbildung 3.15: Werkzeugschemata für Petri-Netz-Werkzeuge

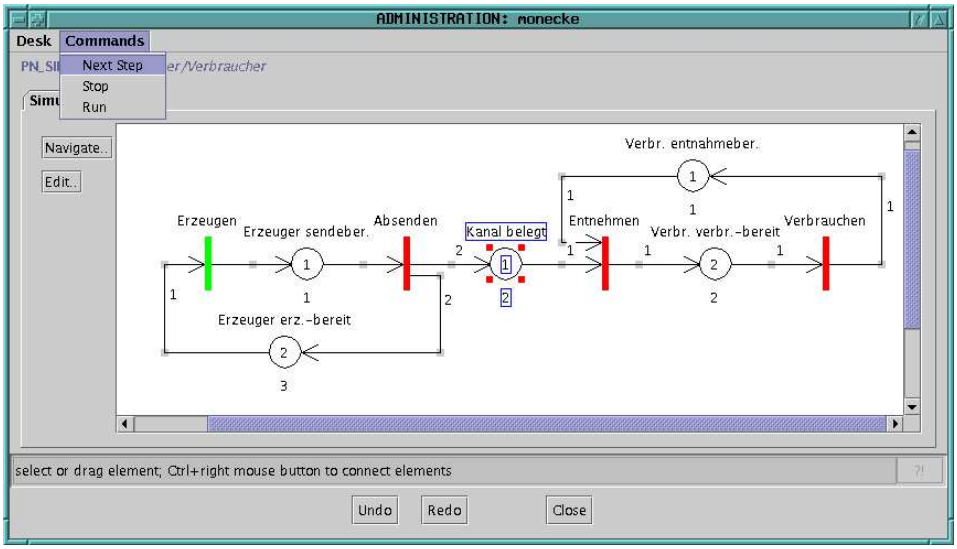


Abbildung 3.16: Anzeige des Petri-Netz-Simulators

– Spezielle Darstellungskomponenten zeigen den Zustand der Elemente des Netzes an – hier die aktuelle Markierung der Plätze und den Zustand der Transitionen. Sie erweitern dazu die Werkzeugkomponenten für die graphische Darstellung im Editor aus Abbildung 3.14.

Simulation. Die Simulation des Petri-Netzes umfaßt

– die Auswertung des Netzes. Dabei muß die Struktur des Netzes und die Markierung bzw. der aktuelle Zustand der Plätze und Transitionen berücksichtigt werden.
– die Überwachung von Änderungen. Diese werden zum einen vom Simulator durchgeführt, zum anderen vom Benutzer, wenn er den Zustand des Netzes oder seine Struktur ändert.
– die Beeinflussung des Zustands durch Änderung von Daten in der Objektbank.

Im Datenmodell des PETRI_SIM-Werkzeugschemas wird der Zustand des Netzes verwaltet. Der Simulator und die Werkzeuge zur Anzeige und Manipulation arbeiten auf der Objektbank und überwachen Änderungen mit dem Benachrichtigungsmechanismus des OMS.

Der Simulator ist als Prozeßagent implementiert – der SimAgent wird im Prozeßmodell aus Abbildung 3.17 der Simulationsaufgabe zugeordnet. Dadurch muß die Simulation nicht als Werkzeugkomponente in die Werkzeuge eingebunden werden, sondern wird von der Prozeßmaschine ausgeführt (mehr zu Prozeßmodellen in Abschnitt 4.2; zur Prozeßmaschine in Abschnitt 5.1).

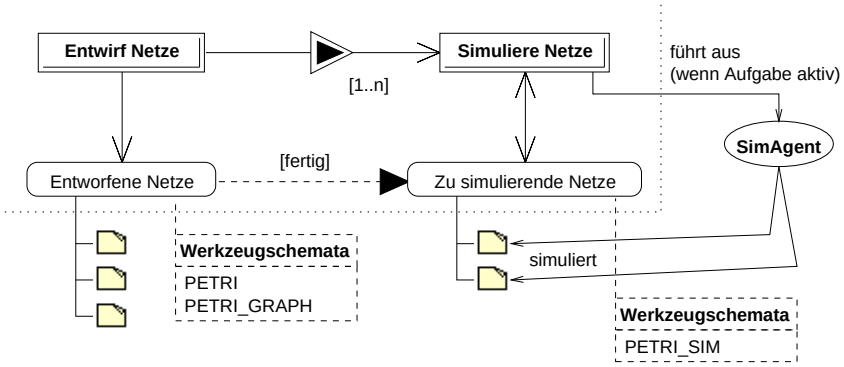


Abbildung 3.17: Prozeßmodell für die Bearbeitung und Simulation von Petri-Netzen

Im Prozeßmodell ist festgelegt, daß die Simulation in einer Simulationsaufgabe ausgeführt wird (Simuliere Netze). Mit dieser Aufgabe kann erst begonnen werden, wenn das Netz fertig bearbeitet und über die Datenflußbeziehung von der Entwurfsaufgabe (Entwerfe Netze) weitergeleitet wurde. Der SimAgent wird mit der Simulationsaufgabe aktiviert und überwacht automatisch die zugeordneten Dokumente.

Das Beispiel zeigt, wie Werkzeuge in *PI-SET* flexibel erweitert werden können, um auch spezielle Anforderungen eines Dokumenttyps zu berücksichtigen. Dies gilt auch für die Prozeßmaschine, die hier um dokumentspezifische Funktionen erweitert wird und mit dem Werkzeug über die Objektbank kommuniziert. Die folgenden Abschnitte erläutern, welche Mechanismen der Werkzeug-Konstruktionsansatz *genform* bereitstellt, um Werkzeuge mit den beschriebenen Eigenschaften zu bauen.

3.1.6 Überblick über den Werkzeugbau mit *genform*

genform besteht aus einem Framework mit Werkzeugkomponenten und aus Meta-Werkzeugen, die den Werkzeugentwickler bei seiner Arbeit unterstützen. Mit *genform* können (textuelle) datenbankbasierte Anwendungen [206] und (graphische) Diagrammeditoren [246] konstruiert werden.

Eine Prozeßunterstützung wird erreicht, indem Entwicklungsprozesse beschrieben und diese Beschreibung von einer Prozeßmaschine interpretiert wird [249]. Planungs- und Management-Werkzeuge werden ebenfalls mit *genform* konstruiert. *genform*-Komponenten kommen auch in der Prozeßmaschine zum Einsatz, wodurch der Implementierungsaufwand verringert und die Integration von Prozeßmaschine und Werkzeugen erleichtert wird.

Auf die Prozeßunterstützung wird erst in Kapitel 4 eingegangen. Abbildung 3.18 zeigt vorab einen Überblick: Zunächst wird zwischen dem Prozeß- und Werkzeugentwurf und der eigentlichen Anwendungsentwicklung unterschieden. Mit *Method Engineering*-Werkzeugen werden

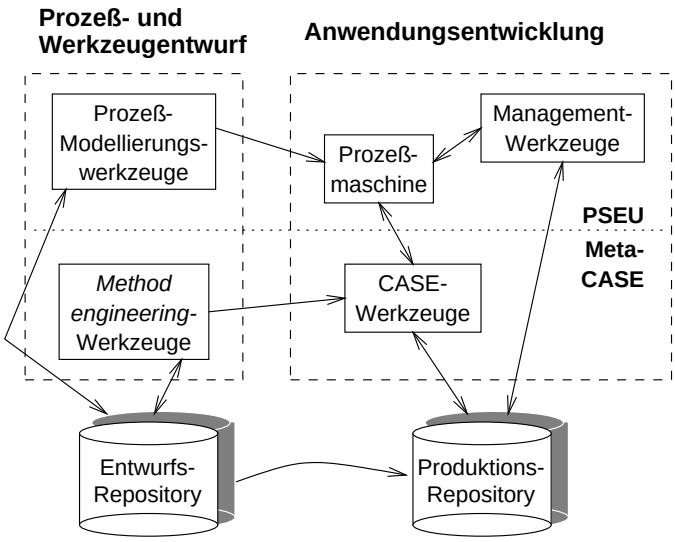


Abbildung 3.18: Werkzeugkonstruktion mit *genform*

CASE-Werkzeuge entworfen, also die verfügbaren Konzepte, Notationen, Konsistenztests und Kommandos festgelegt und implementiert. In *genform* werden Werkzeuge auf verschiedenen Abstraktionsebenen konstruiert: Zum einen durch graphische und textuelle Spezifikationen, zum anderen durch Wiederverwendung von Werkzeugkomponenten sowie durch deren Anpassung und Erweiterung.

Mit den Prozeß-Modellierungswerkzeugen erstellt der Prozeßmodellierer die Architektur des Prozesses, indem er Arbeitsschritte und ihre Beziehungen, sowie die Arten von Dokumenten und ihren Weg durch den Prozeß definiert. Das Ergebnis des Prozeß- und Werkzeugentwurfs ist eine prozeßorientierte Software-Entwicklungsumgebung, die an die Anforderungen eines konkreten Projekts angepaßt ist: Die Entwickler erhalten CASE-Werkzeuge, mit denen die erforderlichen Arten von Dokumenten bearbeitet werden können, und die benötigte Darstellungen, Konsistenztests und Exportkommandos bieten. Planer und Manager erhalten Werkzeuge, mit denen sie das Projekt planen und die Durchführung steuern und überwachen können. Die Werkzeuge müssen hierzu die verschiedenen Arten von Aufgaben und ihre Abhängigkeiten berücksichtigen. CASE- und Managementwerkzeuge interagieren mit einer Prozeßmaschine, die dafür sorgt, daß der Prozeß gemäß Spezifikation abläuft.

Die Daten des Prozeß- und Werkzeugentwurfs werden im *Entwurfsrepository* verwaltet. Es enthält die Werkzeugspezifikationen, das Prozeßmodell und weitere organisatorische Daten. Die Daten des laufenden Prozesses speichert das *Produktionsrepository*. Es enthält die Software-Dokumente und Informationen über den aktuellen Prozeßzustand, also den Zustand von Aufgaben, die verantwortlichen Bearbeiter, aktuelle Aufwände und Kosten.

Durch die horizontale Linie ist angedeutet, welche Forschungsbereiche in *genform* verbunden werden: Die Forschung zu Prozeßmodellierung und PSEU beschäftigt sich mit der Beschreibung von Prozessen und ihrer Ausführung. Die Anpassung und Integration der Werkzeuge ist hier aber meist unzureichend gelöst. Meta-CASE-Systeme zielen auf den Bau methodenspezifischer Werkzeuge, lassen aber meist die Prozesse außer acht.

Beim Bau von CASE-Werkzeugen für eine gegebene Methode müssen zunächst folgende

Fragen geklärt werden:

- Welche Typen von Dokumenten müssen verwaltet und bearbeitet werden, und welche Struktur besitzen diese?
- Welche Rollen existieren im Projekt, welche Organisationsstruktur und Verantwortlichkeiten gibt es?
- Welche Werkzeuge werden von den verschiedenen Rollen zur Bearbeitung benötigt, welche Eigenschaften sollen sie besitzen, welche Kommandos und Darstellungen anbieten?
- Welche Aufgaben werden ausgeführt, wie werden Aufgaben verfeinert und welche Abhängigkeiten existieren zwischen Aufgaben?
- Welche Rollen nutzen welche Werkzeuge bei der Bearbeitung welcher Typen von Dokumenten?

Die Fragen lassen sich nur gemeinsam beantworten, da die Werkzeuge auf den Dokumenten arbeiten und folglich (meist mehrere) Darstellungen für die Dokumente, Navigationsmöglichkeiten in der Dokumentstruktur, Kommandos zum Erzeugen und Löschen von Einträgen und Beziehungen, sowie dokumentspezifische Prüf- und Transformationskommandos anbieten müssen. Durch die Zuordnung von Dokumenttypen und Werkzeugen wird genauer spezifiziert, was in einer Aufgaben wie bearbeitet wird. Durch Anforderungen an den Zustand der Dokumente wird der Prozeßfortschritt an die Weiterentwicklung der Dokumente geknüpft.

3.2 Werkzeugschemata

In *genform* werden das Datenmodell der Dokumente und die Eigenschaften der zugehörigen Werkzeuge in *Werkzeugschemata* (WZS) definiert [246]. Ein Werkzeugschema enthält

- das *Datenmodell* der Dokumente, die das Werkzeug bearbeitet. Das Datenmodell besteht aus Objekt- und Beziehungstypen (*Linktypen*), die beide Attribute besitzen können.
- eine Menge von *Werkzeugparametern*, mit denen die Eigenschaften des Werkzeugs festgelegt werden. Technisch gesehen sind Werkzeugparameter Schlüssel-Wert-Paare, die den Typen im Datenbankschema (Objekt-, Beziehungstypen, Attribute) zugeordnet sind. Die Werkzeuge können zur Laufzeit auf die Parameter zugreifen und ihre Werte auslesen. Durch Werkzeugparameter kann das Aussehen von Formularen und die graphische Darstellung von Dokumenten beeinflußt oder die Zusammensetzung der Werkzeuge aus Werkzeugkomponenten gesteuert werden.
- *Benutztbeziehungen* zu anderen WZS, mit denen einzelne Werkzeuge zu einer Umgebung verknüpft werden. Dem Benutzer können sich solche Beziehungen als Menüeinträge zum Starten eines Werkzeugs präsentieren.

Durch *Import-Beziehungen* zwischen WZS können Typdefinitionen in andere WZS übernommen und dort erweitert werden. Beim Import eines Typs werden auch die zugehörigen Werkzeugparameter importiert. So können leicht verschiedene Varianten eines Werkzeugs konstruiert werden, ohne Werkzeugteile redundant spezifizieren oder implementieren

zu müssen[2].

3.2.1 Feingranulare Modellierung von Dokumenten

Die in den WZS enthaltenen Datenmodelle beschreiben die Dokumente *feingranular*. Ein OOA-Dokument wird also nicht als ein Objekttyp modelliert, in dessen Instanzen der Dokumentinhalt als *long field* gespeichert wird. Vielmehr wird die Dokumentstruktur rekursiv durch Objekt- und Beziehungstypen beschrieben. Die Attribute von Objekt- und Beziehungstypen enthalten die Eigenschaften von Dokumenteinträgen und Beziehungen, wie Name, Beschreibung oder Position im Diagramm. In Abbildung 3.19 ist das bereits aus der Einleitung bekannte Werkzeugschema für einen graphischen Klassendiagramm-Editor nochmals dargestellt. Es werden zwei Arten von Beziehungen unterschieden:

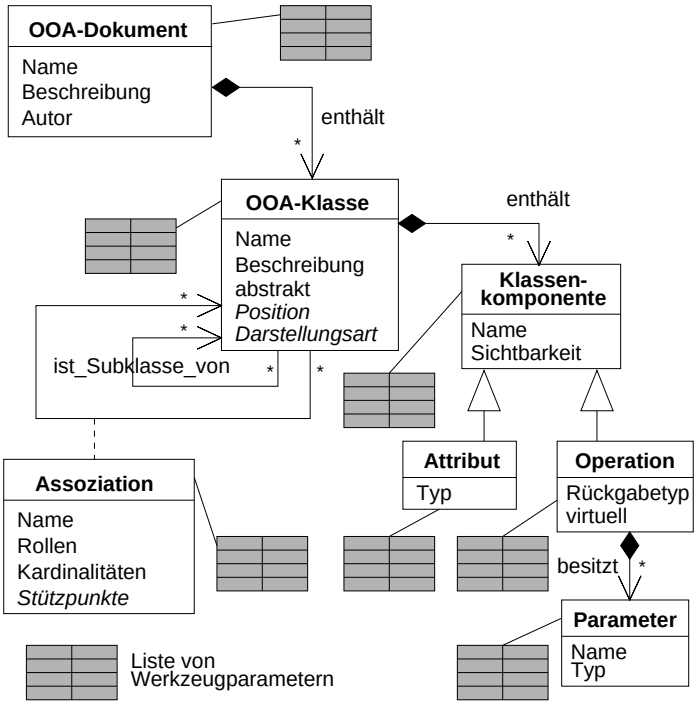


Abbildung 3.19: Vereinfachtes Werkzeugschema für einen Klassendiagramm-Editor

- *Kompositionsbeziehungen* verbinden ein Objekt mit seinen Komponenten – im Beispiel eine Klasse mit ihren Attributen und Operationen. Kompositionsbeziehungen werden durch eine gefüllte Raute gekennzeichnet.
- *Assoziationen* stellen Verweise auf Objekte dar – etwa der Verweis von einer Klasse auf ihre Oberklasse.

Die kursiv dargestellten Attribute speichern Informationen über die graphische Anordnung von Klassen und Beziehungen. Sie sind damit nur für den graphischen Editor relevant,

[2] Dies ist analog zum Sichtenkonzept von Datenbanken, wobei die einzelnen WZS externen Sichtenschemata entsprechen. Die Vereinigung der Typdefinitionen aller WZS bildet dann das konzeptionelle Schema. Tatsächlich wird zur Implementierung der WZS das Sichtenkonzept des Repositories genutzt.

während ein textueller Klassenbrowser auf diese Informationen nicht zugreift. Man kann das WZS des graphischen Editors also als Erweiterung des WZS für den textuellen Browser um zusätzliche Typen und Werkzeugparameter betrachten. Durch letztere werden z.B. den Dokumenteinträgen Werkzeugkomponenten zur graphischen Darstellung zugeordnet. Die Struktur von Klassendiagrammen wird also nur einmal in einem WZS spezifiziert. Andere WZS erweitern dieses zur Beschreibung von graphischen und textuellen Werkzeugen, enthalten also nur die zusätzlichen Merkmale dieser Werkzeuge.

Zu den WZS für den Klassenbrowser und den graphischen Editor gibt es korrespondierende H-PCTE-Schemadefinitionen (SDS), in denen das Datenmodell der Dokumente beschrieben ist. Den Typen in den SDS sind Werkzeugparameter zugeordnet. Das Arbeitsschema des Werkzeugs bestimmt seine aktuelle Sicht: Die Schemaverwaltung des Repositories wird also direkt zur Verwaltung der WZS genutzt und muß nicht in *genform* implementiert werden.

3.2.2 M2-Modell

In Abbildung 3.20 ist das Meta-Meta-Modell (M2-Modell) dargestellt, mit dem in *genform* Werkzeuge spezifiziert werden. Das M2-Modell beschreibt, wie Meta-Modelle (also Modelle für Dokumente) in *genform* erzeugt werden (vgl. auch Abbildung 2.2 auf Seite 48) und nutzt dazu eine Teilmenge der Modellierungskonzepte aus dem PCTE-Standard [317]:

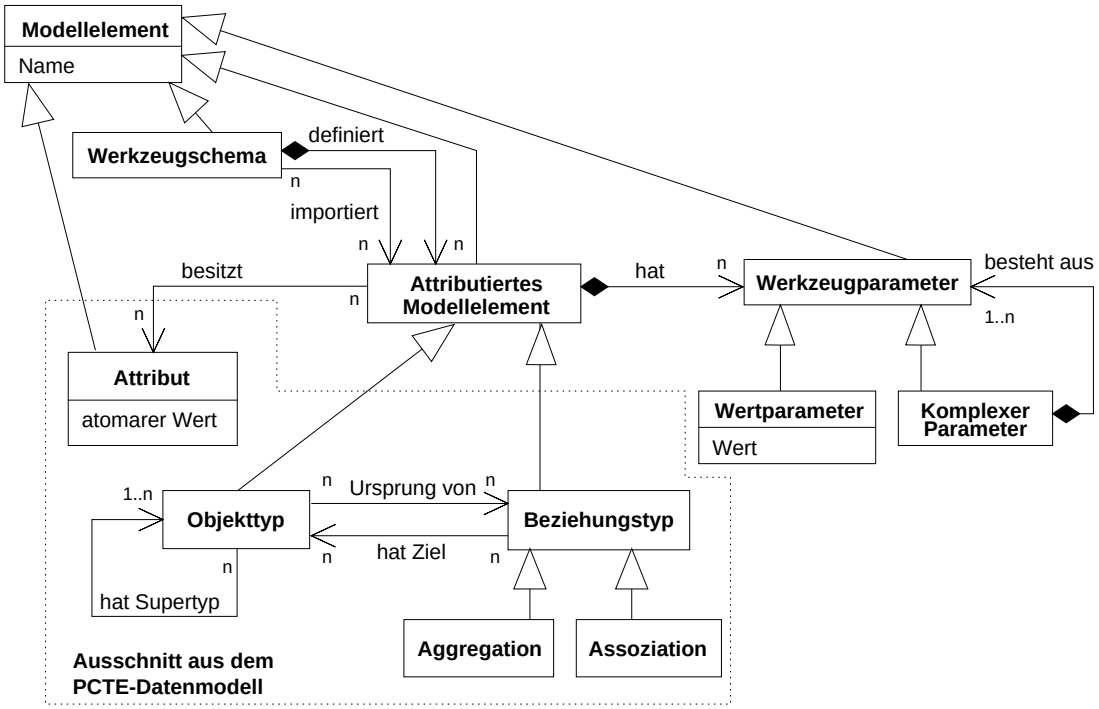


Abbildung 3.20: M2-Modell von *genform*-Dokumenten

– *Objekttypen* repräsentieren Dokumenteinträge. Sie können atomare Attribute besitzen, die die Eigenschaften der Einträge speichern. Objekttypen haben mindestens den Supertyp *object* und können von weiteren Objekttypen Eigenschaften erben. Außerdem sind sie Ursprung und Ziel von beliebig vielen Beziehungstypen.

- *Kompositionsbeziehungen* spannen komplexe Objekte auf. Sie werden durch Linktypen der Kategorie *composition* modelliert und können Nicht-Schlüsselattribute besitzen. Schlüsselattribute werden lediglich zur Unterscheidung der ausgehenden Links verwendet – sie sind nicht Teil des Dokumentdatenmodells. Die Regeln für die Verwendung von Schlüsselattributen müssen beim Entwurf von Datenbankschemata und bei der Implementierung der generischen Komponenten berücksichtigt werden.
- *Assoziationen* bilden Verweise zwischen Einträgen. Für sie werden Linktypen der Kategorie *reference* verwendet, die Nicht-Schlüsselattribute besitzen können.
- *Werkzeugparameter* sind Objekttypen, Beziehungstypen und Attributen zugeordnet. Neben einfachen Werkzeugparametern (*Wertparametern*), die atomare Werte enthalten, gibt es *komplexe Parameter*, die wiederum Werkzeugparameter enthalten.
- Ein *Werkzeugschema* enthält eine Menge von Objekttypen, Beziehungstypen und Attributen mit ihren Werkzeugparametern. Ein Typ wird in genau einem Werkzeugschema *definiert* und kann von beliebig vielen WZS *importiert* werden. Ein Werkzeugschema beschreibt also eine *Werkzeugsicht*, die sowohl das Datenmodell als auch die Werkzeugeigenschaften festlegt. Letztere werden über Werkzeugparameter gesteuert.

3.2.3 Werkzeugparameter

Technisch sind Werkzeugparameter Schlüssel-Wert-Paare. Es gibt zwei Arten von Werkzeugparametern: *Wertparameter* enthalten einfache Werte wie Zeichenketten oder ganze Zahlen. *Komplexe Parameter* enthalten eine Menge anderer Werkzeugparameter. Auf diese wird auch über einen Schlüssel zugegriffen, so daß sich ein hierarchischer Namensraum ergibt.

Da die Werkzeugparameter den Typen im Werkzeugschema zugeordnet sind, wird keine externe Ressource zur Verwaltung benötigt. Somit kann auch die Konsistenz zwischen Typdefinitionen und Werkzeugparametern leichter gewahrt werden. Werkzeugparameter erweitern die Typen im Datenmodell des WZS:

- In Typhierarchien werden auch Werkzeugparameter vererbt. Genauer besteht die Menge der Werkzeugparameter eines Typs aus den ihm selbst zugeordneten Werkzeugparametern und den Werkzeugparametern aller Supertypen bis zur Wurzel der Vererbungshierarchie.
- Beim Import von Typdefinitionen werden auch die zugehörigen Werkzeugparameter importiert. Technisch werden im importierenden WZS Typen mit gleichen Eigenschaften erzeugt; diese Eigenschaften beinhalten auch die Werkzeugparameter.

Geerbte oder importierte Werkzeugparameter können auch überschrieben werden: Wird dem Subtyp oder dem importierten Typ ein Werkzeugparameter mit dem gleichen Namen zugeordnet, überschreibt dieser die geerbten/importierten Eigenschaften.

Nutzung der Werkzeugparameter

genform stellt Funktionen zur Verfügung, mit denen Werkzeugparameter definiert, in Werkzeugschemata verwaltet und von den Werkzeugkomponenten ausgelesen werden können. Wie die Werkzeugparameter genutzt werden, hängt von den einzelnen Werkzeugkomponenten ab.

Werkzeugparameter können Eigenschaften festlegen, die in verschiedenen Werkzeugen und Werkzeugkomponenten benötigt werden. Hierzu zählt der Name des Attributs, das zur Identifizierung von Objekten benutzt werden soll, oder die Art, in der eine Liste sortiert wird. Ersteres wird überall da benötigt, wo ein Objekt textuell dargestellt werden soll; die Sortierung ist für Listen-, Tabellen- und Baumdarstellungen relevant.

Andere Eigenschaften werden nur von bestimmten Werkzeugen benötigt – etwa der Name der Werkzeugkomponente, die die Darstellung von Objekten im graphischen Editor implementiert. Abschnitt 3.4 enthält einige Beispiele für die Verwendung von Werkzeugparametern.

Allgemein können Werkzeugparameter folgende Arten von Werten enthalten:

- Werte zur Steuerung des GUI und der Werkzeugeigenschaften, wie die Zeilenzahl eines Eingabefeldes oder die Art, in der Verbindungslinien im graphischen Editor an den Elementen enden.
- Namen von Typen im Datenbankschema: Ein Beispiel sind die Namen von Attributen, die in einer Tabelle angezeigt werden sollen.
- Namen von Werkzeugkomponenten, aus denen das Werkzeug zusammengesetzt wird – etwa Komponenten, die Dokumenteinträge darstellen oder Werkzeugkommandos implementieren.

Konsistenzsicherung

Bei den beiden letzten Punkten muß gewährleistet sein, daß die adressierten Typen bzw. Komponenten auch tatsächlich existieren. Die Konsistenz zwischen Werkzeugparametern und Datenbankschema ist relativ leicht zu prüfen. Schwieriger ist die Prüfung der korrekten Komponentennamen, da sie ein Verzeichnis der Werkzeugkomponenten erfordern würde. Um die Konsistenz des Werkzeugschemas zu gewährleisten, müßte also die Art des Werkzeugparameters verwaltet und mit einem Prüfwerkzeug festgestellt werden, ob die adressierten Typen bzw. Komponenten tatsächlich existieren. Weitere Prüfungen wären möglich, wenn die Eigenschaften von Typen und Komponenten spezifiziert werden könnten, etwa, ob ein Attribut oder ein Objekttyp angegeben werden muß, oder welche Art von Werkzeugkomponente verwendet werden soll. Voraussetzung für diese Prüfungen ist, daß die Werkzeugparameter selbst in einem Modell beschrieben werden. Ein solches Modell existiert zur Zeit nicht.

3.2.4 Interpretation des Werkzeugschemas

Analog zu den Typeigenschaften kann das Werkzeug auch auf die Werkzeugparameter zugreifen, die den Typen in der aktuellen Sicht zugeordnet sind. Die Werkzeugparameter werden nicht in der Meta-Datenbank verwaltet, sondern in einer transienten Schicht oberhalb des OMS. Die Gründe für die gewählte Realisierung sind:

- Der Implementierungsaufwand ist geringer, da keine Funktionen und Werkzeuge zum Schreiben und Lesen der Werkzeugparameter nötig sind: Die Typeigenschaften können mit speziellen Operationen von H-PCTE direkt aus dem Arbeitsschema ermittelt werden; für die Werkzeugparameter müßten solche Funktionen implementiert werden.
- Die Geschwindigkeit beim Auslesen ist höher, auf eine Pufferung kann verzichtet werden.

– Werkzeugparameter können während der Werkzeugentwicklung leichter geändert werden, da keine Aktualisierung in der Meta-Datenbank nötig ist.

Die transiente Schicht wird beim Werkzeugstart erzeugt: Es wird ein Programmteil ausgeführt, in dem die Initialisierungen kodiert sind. Mit geringem Aufwand könnten die Werkzeugparameter auch in einer externen Ressource, etwa einer XML-Datei, verwaltet werden. In diesem Fall wäre allerdings ein passender Parser nötig und der Zeitaufwand für die Initialisierung höher.

Abbildung 3.21 verdeutlicht, wie die *Eigenschaften* der Typen im Datenmodell des Werkzeugs festgelegt und vom Werkzeug interpretiert werden:

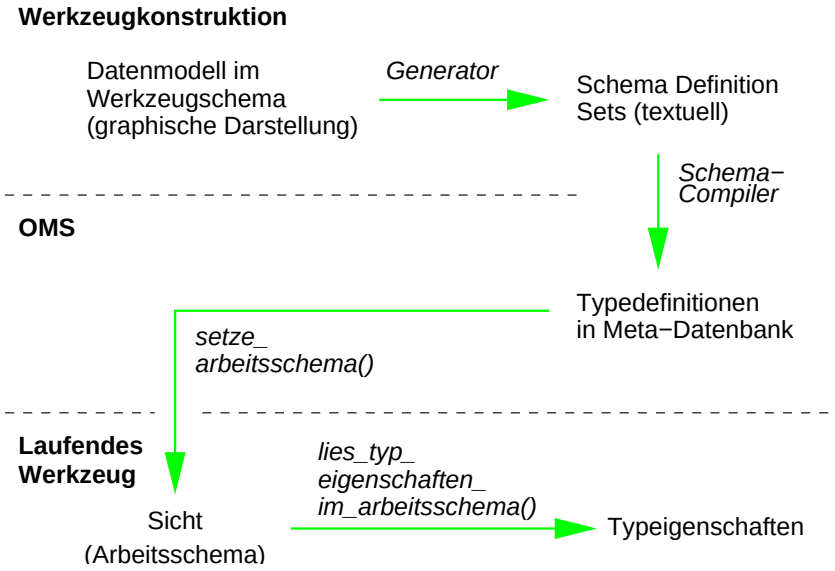


Abbildung 3.21: Definition von Typen und Auslesen der Eigenschaften im Werkzeug

1. Zunächst wird das Datenmodell entworfen. Zur graphischen Darstellung des Datenmodells können verschiedene Notationen verwendet werden, etwa ERA- oder UML-Diagramme. Die in *genform* verwendete Notation lehnt sich an die graphische Notation für SDS an (vgl. Abbildung 2.5 auf Seite 74).
2. Ein Generator erzeugt aus den Diagrammen textuelle SDS-Definitionen. Hierbei fließen bereits Annahmen darüber, wie die Werkzeuge auf den Daten arbeiten, in das SDS ein: Die Regeln, wie Datenbankschemata erzeugt werden, sind im Generator implementiert. Sie legen z.B. fest, welche Attribute an Objekten und Links vorhanden sein müssen (etwa zum Speichern der Position eines Elements in der Zeichenfläche oder innerhalb einer Liste), welche Linkkategorien verwendet werden, und wie die Schlüsselattribute von Links aussehen.
3. Mit einem Schema-Compiler wird die textuelle SDS-Definition in die (Meta-)Datenbank übersetzt (vgl. Abschnitt 2.5.2). Da ein Typ in mehreren SDS vorkommen kann, werden in H-PCTE globale und SDS-spezifische lokale Typeigenschaften unterschieden. Mit letzteren werden auch die Werkzeugeigenschaften sichtenspezifisch gesteuert.

4. Das Arbeitsschema eines Prozesses wird durch eine Folge von SDS festgelegt (Operation *setze_arbeitsschema()*). Es enthält alle Typen der angegebenen SDS, wobei sich die Eigenschaften der Typen aus der Vereinigung der sichtenspezifischen Eigenschaften in den jeweiligen SDS ergeben.
5. Das laufende Werkzeug kann die Eigenschaften der Typen im Arbeitsschema über Schnittstellen des OMS auslesen (*lies_typ_eigenschaften_im_arbeitsschema()*). Die Eigenschaften werden genutzt, z.B. um für alle Attribute eines Objekt- oder Linktyps die Eingabefelder innerhalb eines Formulars zu bestimmen, oder Kommandos zum Erzeugen von Komponenten (Kompositionsbeziehung) oder Beziehungen zwischen Objekten (Assoziation) zu erzeugen.

3.3 Werkzeugkomponenten

Werkzeugschemata legen die Eigenschaften und die Zusammensetzung von Werkzeugen fest. Die Bausteine der Werkzeuge sind die Werkzeugkomponenten aus dem *genform*-Framework.

3.3.1 Zusammensetzung der Werkzeuge

Bei der Zusammensetzung der Werkzeuge ergänzen sich die Interpretation des Datenmodells und der Werkzeugparameter: Jedes Werkzeug wird auf dem Wurzelobjekt eines Dokuments gestartet und ermittelt zunächst den Typ dieses Objekts. Anhand des Typs werden Typeigenschaften und Werkzeugparameter aus dem Werkzeugschema ausgelesen, danach die GUI zur Bearbeitung des Objekts erzeugt.

Vorgegeben sind folgende GUI: Die Eigenschaften von Objekten und Links werden mit Formularen bearbeitet. Ein Formular enthält Eingabefelder für alle (in der aktuellen Sicht enthaltenen) Attribute. Die Art des Eingabefelds hängt vom Typ des Attributs ab – Eingabefelder für Texte, Auswahlfelder für Aufzählungstypen. Ausgehende Beziehungen werden in Listen bearbeitet. Jedes Zielobjekt wird hier durch einen Listeneintrag dargestellt, der mindestens ein Attribut des Zielobjekts oder des Links zum Zielobjekt anzeigt. Listen für Kompositionsbeziehungen und für Assoziationen unterscheiden sich in den Kommandos zum Erzeugen der Beziehung: Bei der Komposition wird die Komponente zusammen mit der Beziehung erzeugt, es muß also ein Kommando zum Erzeugen von Instanzen jedes möglichen Zielobjekttyps vorhanden sein. Bei Assoziationen wird ein existierendes Zielobjekt ausgewählt; hier müssen also gültige Zielobjekte in einer Liste angeboten werden.

Die verschiedenen GUI erscheinen auf Tabulatorkarten im Werkzeugfenster. In Abbildung 3.22 sind zwei Tabulatorkarten eines Werkzeugs dargestellt. Zusätzlich ist ein Ausschnitt aus der Hierarchie von Werkzeugkomponenten skizziert, aus denen das Werkzeug besteht. Um das Aussehen des GUI zu beeinflussen, können an Attributen, Objekttypen und Linktypen Werkzeugparameter definiert werden. Statt der Listendarstellung kann so auch eine Tabellen- oder Baumdarstellung verwendet werden. Allgemein können alle Komponenten im Werkzeug durch andere ersetzt werden. Der gewünschte Typ muß dazu in einem Werkzeugparameter angegeben werden.

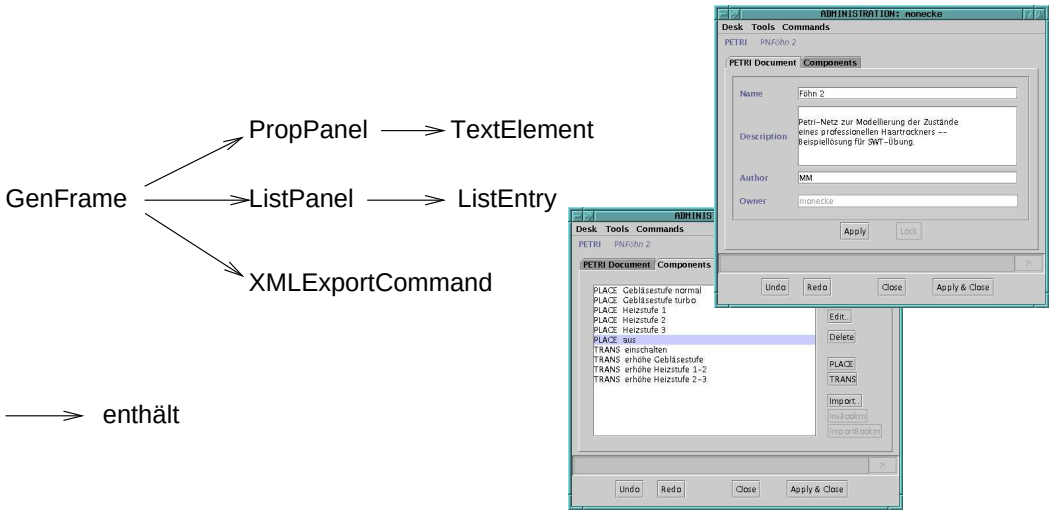


Abbildung 3.22: Werkzeugkomponenten und GUI zu Bearbeitung von Eigenschaften und Beziehungen eines Objekts

Komponenten können 'Steckplätze' besitzen, in die anderen Komponenten einrasten. In Abbildung 3.23 ist skizziert, welche Komponenten in einem graphischen Editor verwendet werden. Das folgende Beispiel verdeutlicht den Ablauf bei der Interpretation des Werkzeugschemas.

Beispiel 3.1 (Erzeugen einer Listendarstellungen) In Abbildung 3.24 sind einige der Komponenten skizziert, wie sie in einer Listendarstellung verwendet werden (vgl. Abbildung 3.22 unteres Fenster). Die dargestellten Quelltextauszüge sollen den Ablauf bei der Interpretation der Werkzeugschemata erläutern. Auf die verwendeten Komponenten wird in den folgenden Abschnitten näher eingegangen.

1. Die Klasse GenFrame implementiert ein generisches Werkzeugfenster. Es ist genau einem Wurzelobjekt zugeordnet, auf das die Objektreferenz root_ref verweist. Das Fenster interpretiert das Werkzeugschema und erzeugt in der Operation makeContent Anzeigefelder (*panels*) auf Tabulatorkarten. Jedes *panel* zeigt einen Aspekt des Objekts, also etwa seine Attribute, die Zielobjekte einer ausgehenden Beziehung oder das GUI eines zugeordneten Werkzeugs.
2. Beim Erzeugen der *panels* für die ausgehenden Beziehungen in GenFrame.createListPanels wird zunächst ein LinkCache erzeugt, in den die ausgehenden Links des Objekts eingelesen werden (mehr zum LinkCache in Abschnitt 3.3.2 auf Seite 123). Instanzen der Klasse AssocType kapseln die Typeigenschaften und bieten Operationen zur Abfrage an (in der Abbildung nicht dargestellt). Soll eine ausgehende Beziehung in einer Listendarstellung angezeigt werden, liefert AssocType.showAsList den Wert *true* zurück – für den Beziehungstyp wird dann ein ListPanel erzeugt. Ob die Beziehung als Liste dargestellt werden soll oder nicht, wird über einen Werkzeugparameter festgelegt, dessen Vorgabewert *true* ist.
3. Die Typeigenschaften werden aus der Meta-Datenbank ermittelt. Bei Anfragen nach Objekttypen kann der Typname oder eine Objektreferenz angegeben werden. Eine Instanz

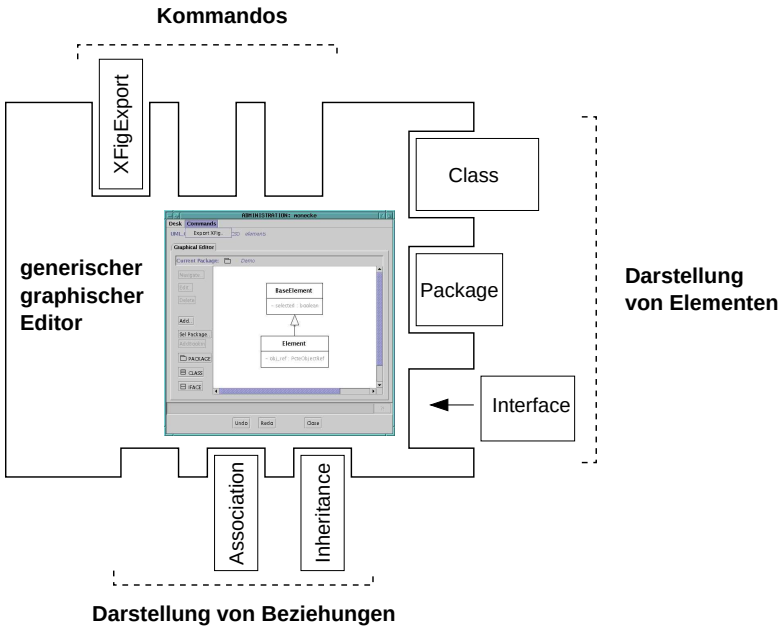


Abbildung 3.23: Zusammensetzung von Werkzeugen

der Klasse ObjType (in der Abbildung nicht dargestellt) enthält die Eigenschaften eines Objekttyps; ObjType.getAssocs liefert die ausgehenden Beziehungstypen.

4. Ein ListPanel zeigt die ausgehenden Links eines Typs an. Die Links und die zugehörigen Zielobjekte werden aus dem LinkCache ausgelesen (LinkCache.getEntries). Genauer besitzt das ListPanel einen Verweis auf den LinkCache des Fensters, der alle ausgehenden Links verwaltet. Für jeden ausgehenden Link wird eine Link-Beschreibung (LinkDescr) erzeugt. Sie enthält den Linknamen, eine Objektreferenz auf das Zielobjekt und weitere Eigenschaften des Links. Für jeden Link erzeugt das ListPanel einen Listeneintrag.

5. Ein Listeneintrag zeigt Informationen über das Zielobjekt an, meist die Werte von Attributen des Zielobjekts oder des Links. Die Komponenten, die als Listeneinträge verwendet werden, sind im Werkzeugschema festgelegt: Einträge vom Typ ListEntry zeigen die Werte von *string*-Attributen durch Komma getrennt an; ParListEntry zeigt das zweite Attribut in Klammern an (etwa für Darstellungen der Form `Diagramm1 (Meier)`); LinkListEntry zeigt die Attribute des Links an. Die Listeneinträge können direkt editiert werden und ermitteln die Werte der einzelnen Attribute mit einem Parser aus der eingegebenen Zeichenkette. Für die Darstellung etwa von Attributen und Operationen einer UML-Klasse werden werkzeugspezifische Eintragskomponenten benötigt, die passende Darstellungen und Parser implementieren.

   Beim Erzeugen von Einträgen in ListPanel.createEntry werden Werkzeugparameter an den Objekttypen der Listeneinträge ausgewertet. Sie bestimmen, welcher Eintragstyp verwendet wird und welche Attribute angezeigt werden sollen. Es wird also zunächst die Eintragsklasse ermittelt (ListPanel.getEntryClass), dann ein Eintrag erzeugt, initialisiert und in die Liste eingefügt. Der Eintrag wiederum ermittelt die anzuzeigenden Attribute, liest ihre Werte ein und meldet Notifizierer zum Überwachen von Änderungen an.

□

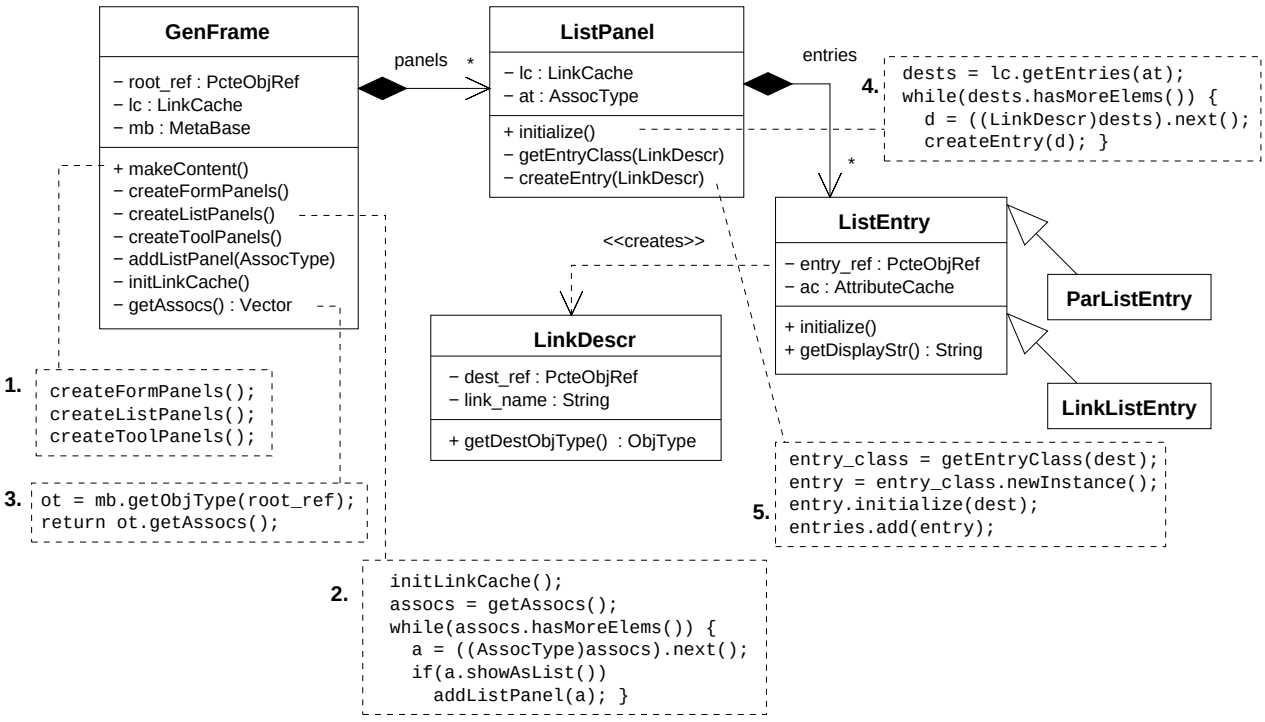


Abbildung 3.24: In der Listendarstellung verwendete Klassen und Auszüge aus dem Quelltext zur Schema-Interpretation

3.3.2 Struktur des *genform*-Frameworks

Die in *genform* enthaltenen Komponenten lassen sich grob in drei Arten unterteilen:

1. *GUI-Komponenten* implementieren generische Benutzungsschnittstellen. Es gibt Komponenten für die Bearbeitung von Dokumenten in textueller und graphischer Darstellung.
2. *Werkzeuglogik-Komponenten* implementieren die Werkzeugfunktionen.
3. *OMS-Komponenten* implementieren eine OMS-Zugriffsschicht, in der Schnittstellenoperationen von H-PCTE so aufbereitet werden, daß sie von den Werkzeugkomponenten einfacher und einheitlich verwendet werden können.

genform enthält zur Zeit etwa 320 Klassen. Abbildung 3.25 zeigt eine Übersicht über die wichtigsten Pakete in *genform*.

view enthält alle GUI-Klassen, außer den Klassen für graphische Editoren; diese befinden sich in graph. Die Subpakete enthalten Komponenten für Fenster (window) und die enthaltenen Anzeigeflächen (panel). Komponenten für Kommandos sind in cmd zusammengefaßt: Sie implementieren die GUI-Aspekte von Kommandos, also ihre Darstellung und die Anzeige von Hilfetexten und Fehlermeldungen. Eine enthaltene Klasse ist FileCmd. Sie implementiert auch einen Dateidialog, mit dem Verzeichnisse oder Dateien ausgewählt werden können, etwa beim Export von Grafikdateien. Die Unterpakete von panel enthalten jeweils Komponenten für verschiedene Arten generischer GUI für textuelle Darstellungen:

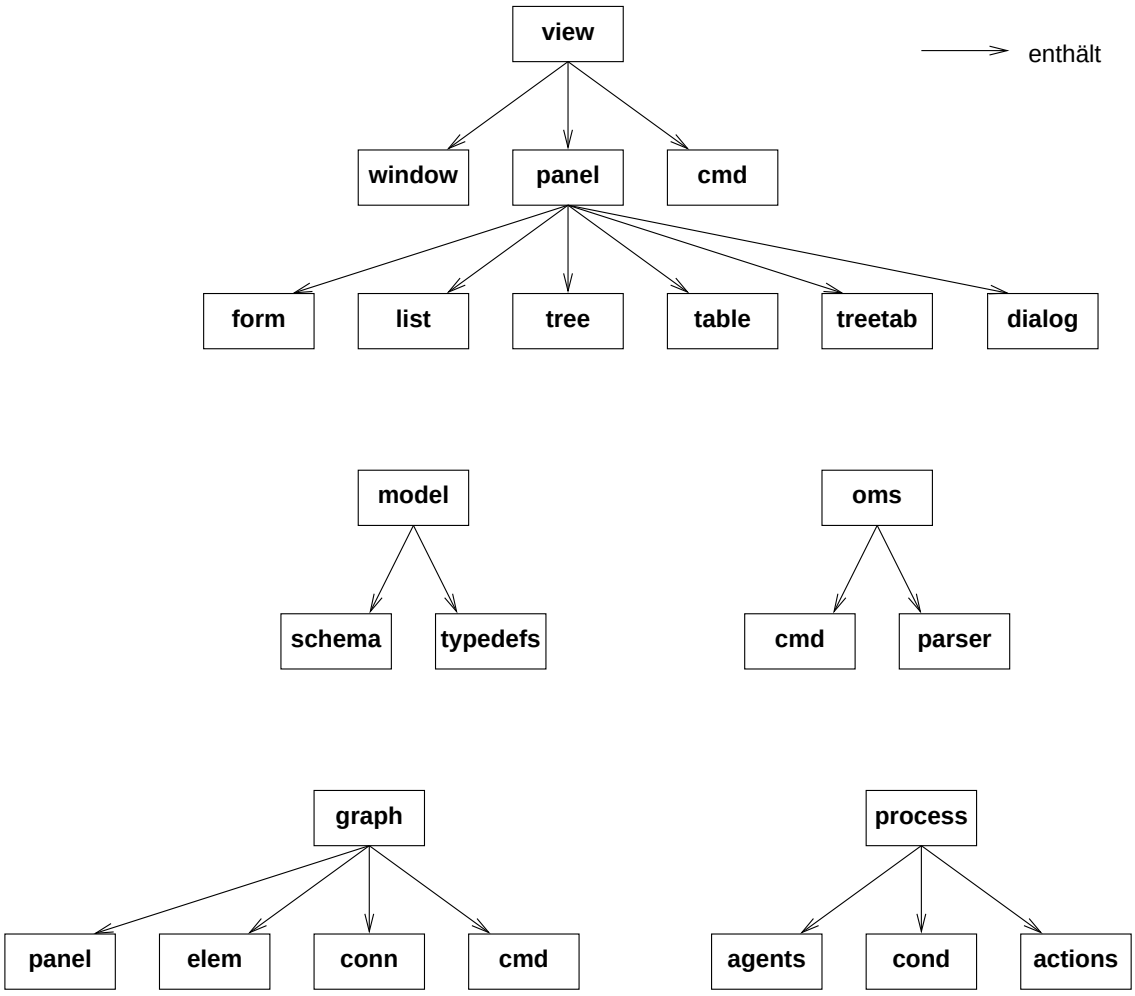


Abbildung 3.25: Übersicht über die *genform*-Pakete

- *Formulare* mit verschiedenen Arten von Eingabefeldern. Die Eingabefelder sind typabhängig (textuelles Eingabefeld für Strings, Ankreuzfelder für boolesche Werte) und werden automatisch anhand der Attribute im Datenmodell erzeugt. Die Eigenschaften des Formulars und der einzelnen Felder können durch Werkzeugparameter gesteuert werden.
- *Listendarstellungen* mit verschiedenen Arten von Listeneinträgen. Eine Liste zeigt eine Menge von Objekten an, die über Links eines Typs vom Wurzelobjekt aus erreichbar sind. Die Listeneinträge können ein oder mehrere Attribute der Zielobjekte anzeigen und das Editieren der Attributwerte erlauben. Werden mehrere Attribute angezeigt, müssen die Ausgaben formatiert und eingegebene Texte in die einzelnen Attributwerte zerlegt werden.
- *Baumdarstellungen* zeigen Hierarchien von Objekten an. Jedes Objekt wird durch eine Folge von Attributwerten dargestellt, die direkt in der Baumdarstellung editiert werden können. Jeder angezeigte Knoten bietet Kommandos zum Erzeugen und Löschen von untergeordneten Knoten an.
- *Tabellendarstellungen* zeigen die Attribute einer Menge von Objekten tabellarisch

an. Die Funktion entspricht der Listendarstellung, allerdings werden die Attributwerte auf die Tabellenspalten verteilt. Mit der verwendeten Tabellenkomponente können auch Spalten umsortiert und in ihrer Breite verändert werden.

- *Baum-Tabellendarstellungen* zeigen eine Baumstruktur an, wobei jeder Knoten im Baum durch eine Tabellenzeile repräsentiert wird. Die Funktionen entsprechen also denen der Baum- und der Tabellenkomponente.
- Das Unterpaket dialog enthält Komponenten für Standard-Dialoge.

model enthält Komponenten, mit denen die Meta-Datenbank einer *genform*-Anwendung realisiert wird – sie bilden also die Basis für die Schema-Interpretation. Es ist unterteilt in Klassen, mit denen Werkzeugschemata erzeugt werden, und Klassen, die Typdefinitionen in der Meta-Datenbank repräsentieren.

oms enthält die Komponenten der OMS-Zugriffsschicht. Sie implementieren generische Funktionen zum Zugriff auf die Daten im OMS, zum Prüfen von Zugriffsrechten und Sperren und zum Überwachen von Änderungen. Das Unterpaket cmd enthält OMS-Kommandos, etwa zum Erzeugen und Löschen von Objekten und Links. Die Kommandos rufen die passenden API-Operationen auf und reagieren auf Fehler. Die Kommandos in view.cmd stützen sich hierauf ab.

graph enthält Komponenten für das GUI generischer graphischer Editoren. Dazu zählen Anzeigeflächen (panel) und Komponenten für die Darstellung von Elementen (elem) und Beziehungen (conn). In cmd sind, analog zu view.cmd, die Kommandos für graphische Editoren enthalten, etwa zum Erzeugen von Beziehungen in der Zeichenfläche oder zum Verschieben von Elementen (siehe auch Beispiel 3.3).

process enthält Komponenten, mit denen eine Prozeßmaschine gebaut werden kann (mehr dazu in Abschnitt 5.1); unterteilt in Komponenten für Prozeßagenten (agents) und Komponenten, mit denen Bedingungen und Aktionen eines Zustandsautomaten implementiert werden (cond, actions).

Im folgenden wird auf die verschiedenen Arten von Komponenten genauer eingegangen.

GUI-Komponenten

Die GUI-Komponenten dienen zur Darstellung und Manipulation von Daten im OMS. Sie basieren auf den Komponenten des verwendeten GUI-Frameworks und ergänzen sie um die OMS-Anbindung und Funktionen, die in der OMS-orientierten Architektur benötigt werden:

- Die Daten werden aus dem OMS in die Datenstrukturen der GUI-Komponente geladen: Die Listendarstellung verwendet ein ListModel, Baumdarstellungen werden aus TreeNode-Instanzen aufgebaut (vgl. Beispiel 3.2).
- Änderungen werden direkt im OMS durchgeführt. Sie werden zum einen über Menüeinträge oder Funktionsknöpfe aufgerufen. Zum anderen ermöglichen auch manche GUI-Komponenten die direkte Manipulation. Ein Beispiel sind Baumknoten, in denen die Knotenbeschriftung editiert werden kann. In diesem Fall muß die Komponente um die Aufrufe der OMS-Operationen erweitert werden.

– Relevante Ressourcen werden vom Benachrichtigungsmechanismus des OMS überwacht und bei einer Änderung automatisch die GUI-Datenstrukturen und die Darstellung aktualisiert.
– Rechte und Sperren an den Ressourcen im OMS werden berücksichtigt, also beim Initialisieren der Komponente die Rechte- und Sperrsituation geprüft und die Interaktionsmöglichkeiten angepaßt. Eingeschränkte Interaktionsmöglichkeiten müssen auch im GUI sichtbar sein, etwa durch 'Ausgrauen' von Menüeinträgen und Eingabefeldern. Die Rechte- und Sperrsituation an Ressourcen ist nicht statisch, sondern kann durch parallel laufende H-PCTE-Prozesse beeinflußt werden. Die Rechte- und Sperrsituation wird daher mit Notifizierern überwacht und bei einer Änderung das GUI angepaßt, etwa durch Aktivieren oder Deaktivieren von Kommandos und Eingabefeldern.

Beispiel 3.2 (Baumdarstellung) In Abbildung 3.26 ist skizziert, welche Komponenten in einer Baumdarstellung verwendet werden:

– GenFrame implementiert das Werkzeugfenster. Instanzen enthalten eine Objektreferenz auf das Wurzelobjekt des bearbeiteten Dokuments. Ein GenFrame interpretiert das Werkzeugschema und erzeugt die benötigten GUI auf Tabulatorkarten. Die gewünschte Baumdarstellung muß über Werkzeugparameter spezifiziert werden; vorgegeben ist die Listendarstellung.
– GenTreePanel erzeugt eine Baumdarstellung des Dokuments. Es erweitert die Basisklasse TreePanel, da es das Werkzeugschema interpretiert und die Baumstruktur selbständig aufbaut. Das TreePanel nutzt die *Swing*-Komponente JTree zur Darstellung des Baums. Die Knoten des Baums implementieren die Schnittstelle TreeNode.
– Die Dokumenteinträge werden durch Instanzen der Klasse RefNode implementiert: Ein RefNode enthält eine Objektreferenz auf den Dokumenteintrag; außerdem Operationen zur Darstellung des Eintrags, zum Erzeugen von Kommandos (im *Popup*-Menü oder in der Werkzeugleiste) sowie die Rückrufoperation pcteNotify. Letztere wird vom OMS aufgerufen, wenn der Knoten über eine Änderung in der Objektbank informiert werden soll. Mit einem RefNode können die angezeigten Attribute auch direkt editiert werden. Zur Freigabe des Editors wird die Rechte- und Sperrsituation geprüft und mit Notifizierern überwacht.

□

Beispiel 3.3 (Graphischer Editor) Abbildung 3.27 zeigt die Komponenten, die in einem graphischen Editor für Klassendiagramme verwendet werden.

– Die Klassen für das GUI des graphischen Editors befinden sich im Paket genform.graph: Das EditorPanel enthält die generische Zeichenfläche (DrawArea), auf der die Dokumente dargestellt und bearbeitet werden. Die Zeichenfläche verwaltet Einträge und Beziehungen im Dokument. Dokumenteinträge werden durch Subklassen von Element implementiert; Beziehungen durch Subklassen von Connection.
– Werkzeugspezifische Komponenten erweitern die *genform*-Komponenten. Beispiele sind Class und Association, die die graphische Darstellung von Klassen und Assoziationen im

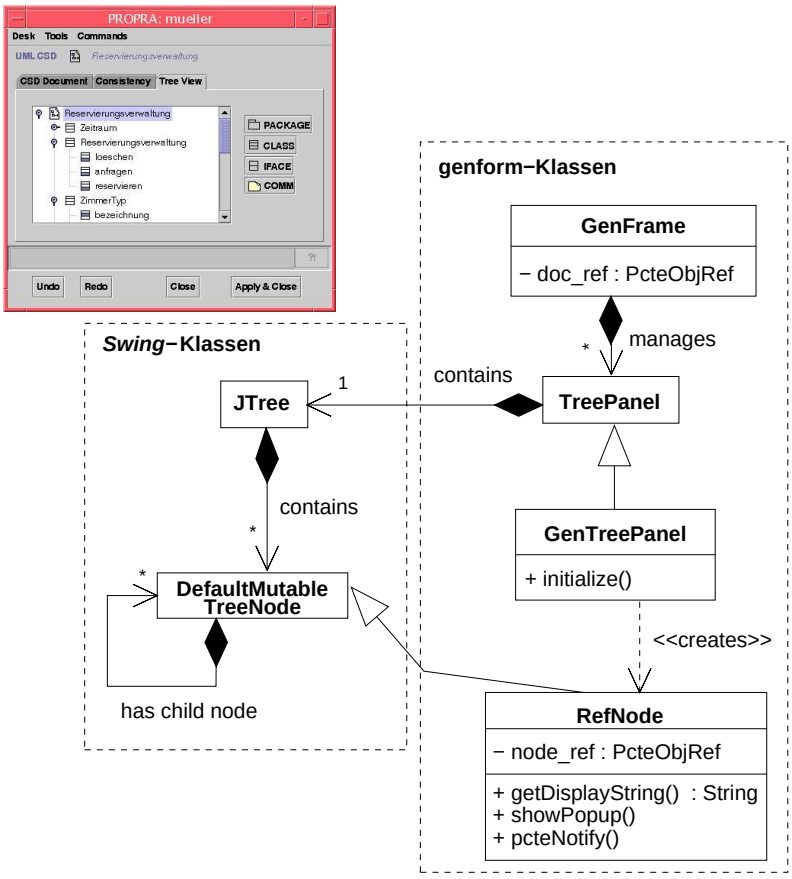


Abbildung 3.26: Klassen zur Realisierung einer Baumdarstellung und resultierendes GUI

Klassendiagramm implementieren. Die werkzeugspezifischen Klassen befinden sich in einem werkzeug- oder anwendungsspezifischen Paket (hier piset.uml_csd, das Paket mit den Komponenten für einen Klassendiagramm-Editor in *PI-SET*).

– Die Zeichenfläche erzeugt die verfügbaren Editierkommandos selbst durch Interpretation des Werkzeugschemas. Die Kommandoklassen sind Subklassen von GAction. Sie befinden sich in genform.graph, weil sie an den graphischen Editor angepaßt sind. Beispiele sind das Kommando zum Erzeugen von Dokumenteinträgen (CreateCmd), von dem je eins pro Eintragstyp instantiiert wird, und das Kommando zum Editieren der Eigenschaften von Einträgen und Beziehungen in einem separaten Fenster (EditCmd).

□

Werkzeuglogik-Komponenten

Die Werkzeuglogik-Komponenten fassen Funktionen zusammen, die in verschiedenen GUI verwendet werden. Hier werden zwei Arten solcher Komponenten vorgestellt: Komponenten, die Werkzeugkommandos implementieren und Komponenten, mit denen der Zustand des GUI gesteuert wird.

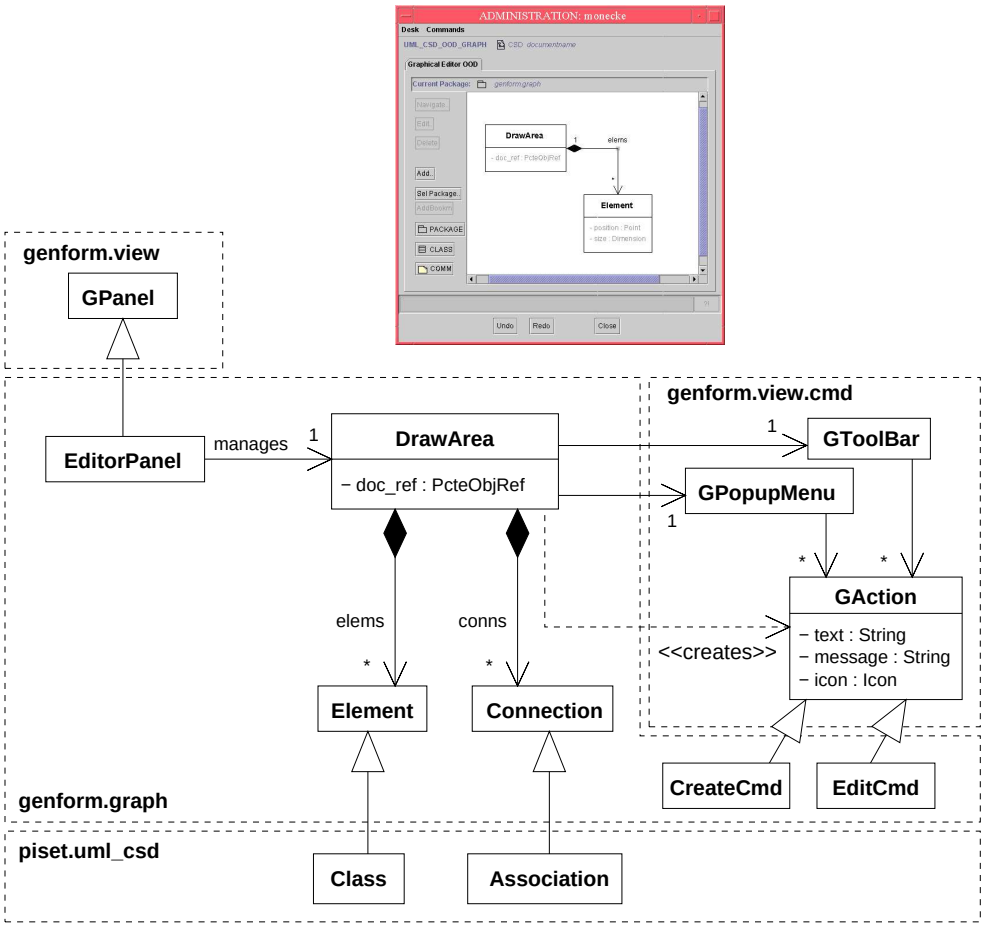


Abbildung 3.27: Klassen in einem graphischen Editor und resultierendes GUI

Werkzeugkommandos. Die Werkzeug-GUI ermöglichen neben der Darstellung der Dokumente natürlich auch deren Manipulation. Beispiele sind das Editieren eines Textes, das Verschieben eines Elements im Diagramm und das Löschen eines Eintrags. In einer herkömmlichen Werkzeugarchitektur werden die Änderungsoperationen von der transienten Dokumentverwaltung durchgeführt. Diese hängt stark von der Repräsentation des Dokuments ab, da eine Baumstruktur anders verwaltet und manipuliert wird als eine Liste. Andererseits unterscheidet sich die Semantik eines Löschkommandos in einer Listen- und einer Baumdarstellung nicht voneinander, so daß auch die Implementierung der Kommandos weitgehend gleich sein sollte.

In der OMS-orientierten Werkzeugarchitektur werden Werkzeugkommandos direkt in der Objektbank durchgeführt: Das Löschen eines Listeneintrags und eines Teilbaums wird also auf die gleichen Datenbank-Operationen abgebildet, hier das Löschen eines komplexen Objekts. Die Darstellung wird dann aufgrund einer Änderungsnachricht vom OMS aktualisiert (vgl. Abschnitt 3.1.1).

Somit liegt es nahe, die Werkzeugkommandos unabhängig vom GUI zu implementieren und in der Werkzeuglogik-Schicht zusammenzufassen. Ein Werkzeugkommando enthält

– den Aufruf von Funktionen aus der OMS-Zugriffsschicht, um das Dokument in der Objektbank zu ändern.

– den Aufruf von Funktionen aus dem GUI, um nötige Eingaben vom Benutzer zu erhalten oder Hinweise und Fehlermeldungen anzuzeigen.

Beispiel 3.4 (Erzeugen von Beziehungen in einer textuellen Darstellung) *genform* enthält ein generisches Kommando zum Erzeugen von Beziehungen zwischen Objekten. Das Kommando ermöglicht die Auswahl des Zielobjekts der Beziehung und erzeugt dann einen Link in der Objektbank. Anhand des SDS für OOA-Diagramme aus Abbildung 3.28 soll die Funktion des Kommandos erläutert werden:

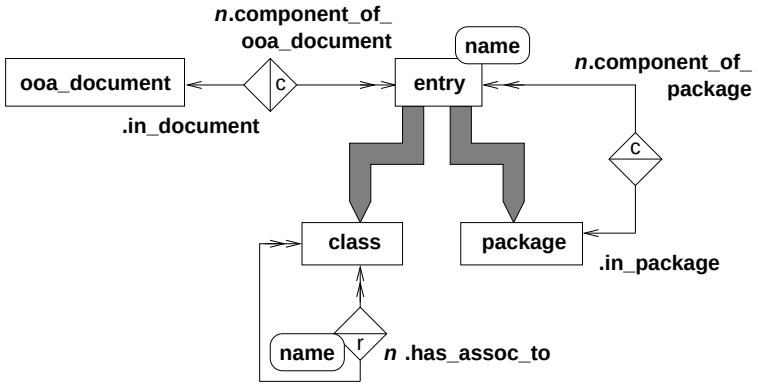


Abbildung 3.28: Ausschnitt aus einem SDS für OOA-Diagramme

Im SDS werden Assoziationen durch den Linktyp has_assoc_to repräsentiert. Abbildung 3.29 links zeigt eine textuelle Darstellung einer Paketstruktur. Die ausgehenden Assoziationen einer Klasse werden in einer Liste dargestellt (Abbildung 3.29 rechts oben). Zum Erzeugen einer Assoziation muß zunächst die Zielklasse ausgewählt werden; diese kann sich in einem beliebigen Paket befinden. Zur Auswahl wird daher die gesamte Paketstruktur mit dem Dokument als Wurzel in einer Baumdarstellung angeboten (Abbildung 3.29 rechts unten). Die Darstellung enthält allerdings nur die zur Navigation und Auswahl relevanten Einträge – hier also keine Schnittstellen oder Kommentare.

Das Kommando nutzt die im vorigen Abschnitt vorgestellten GUI-Klassen zur Realisierung der Benutzungsschnittstelle. Es interpretiert zum einen das Datenmodell, kennt also die Typen möglicher Zielobjekte (class) der Beziehung und den Linktyp (has_assoc_to). Zum anderen wird das Kommando über Werkzeugparameter wie folgt konfiguriert:

– Das Zielobjekt wird in einer Baumstruktur ausgewählt. Für eine 'flache' Objektmenge würde eine Liste verwendet.
– Es müssen die Linktypen spezifiziert werden, die den Baum aufspannen. Hier sind dies die Linktypen component_of_ooa_document und component_of_package.
– Der Pfad zum Wurzelobjekt des Verzeichnisses, in dem sich die möglichen Zielobjekte befinden, wird über einen regulären Pfadausdruck spezifiziert. Ausgehend vom Klassenobjekt wird durch die Paketstruktur bis hinauf zum Dokument navigiert:
  .in_package*/.in_document
– Es sind nur Klassen als Zielobjekte erlaubt, keine Pakete. Letztere erscheinen zwar im Baum, können aber nicht als Zielobjekt ausgewählt werden.

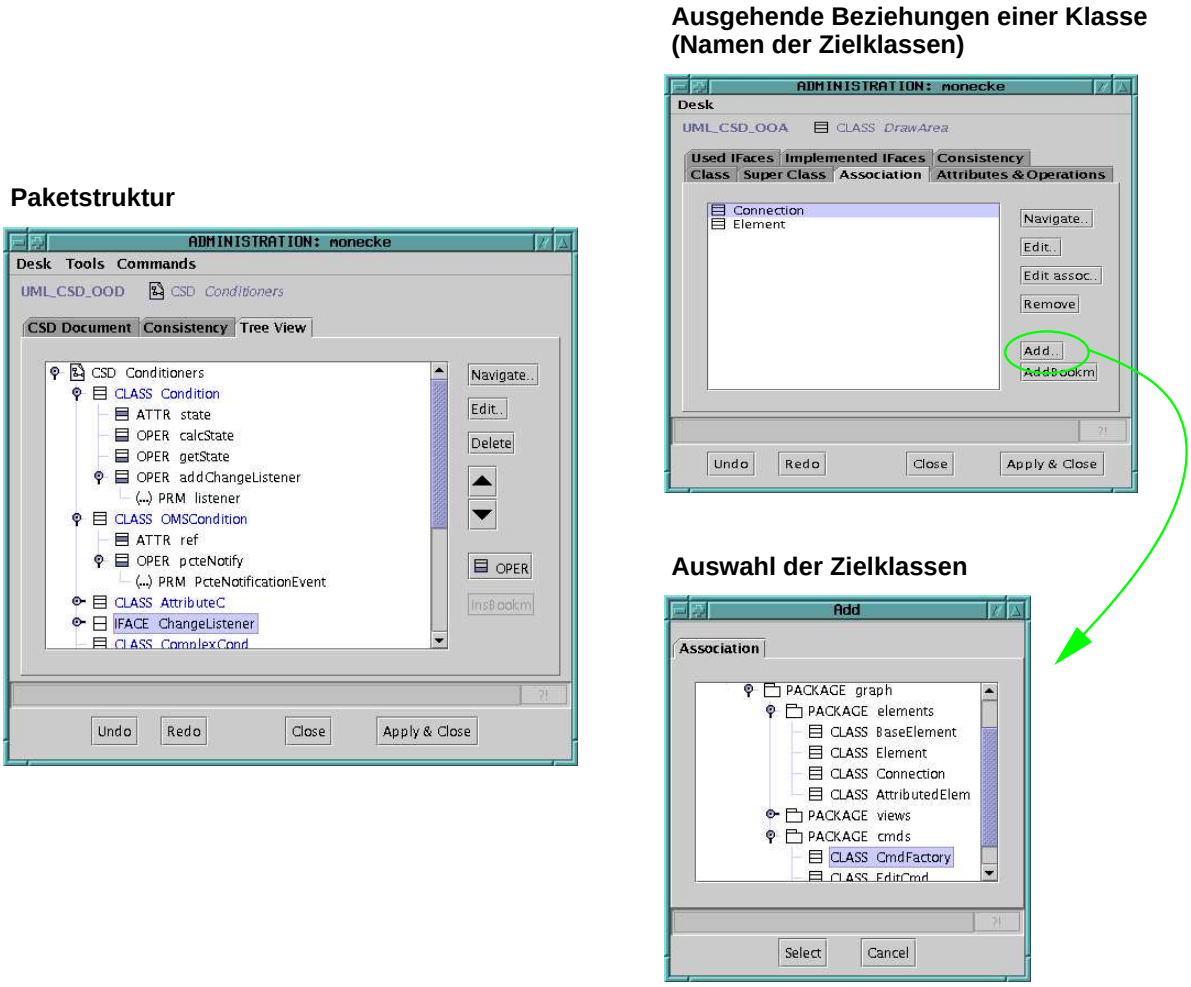


Abbildung 3.29: Erzeugen einer Assoziation zwischen Klassen

– Rekursive Beziehungen sind hier erlaubt, da eine Klasse eine Assoziation zu sich selbst haben kann – die Ausgangsklasse erscheint also auch in der Auswahl. Beim Erzeugen einer Vererbungsbeziehung gilt dies nicht.
– Duplikate sind hier erlaubt, da eine Klasse mehrere Assoziationen zur gleichen Klasse haben kann – Klassen, zu denen bereits eine Assoziation besteht, erscheinen also auch in der Auswahl. Beim Erzeugen einer Vererbungsbeziehung gilt auch dies nicht.

□

Bedingungen. Welche Daten im GUI angezeigt werden, welche Daten editierbar sind, und welche Kommandos ausgeführt werden können, hängt von verschiedenen Faktoren ab:

– Vom Zustand des GUI: Das Löschkommando sollte nur aktiv sein, wenn mindestens ein Listeneintrag selektiert ist.
– Vom Zustand des bearbeiteten Dokuments: Ein Transformationskommando sollte nur aktiv sein, wenn das Dokument auch Einträge enthält.

– Vom Zustand der Ressourcen in der Objektbank: Ein Eingabefeld sollte nur editierbar sein, wenn das Schreiben des zugehörigen Attributs nicht durch Typrechte, Zugriffsrechte oder Sperren verhindert wird.

Allgemein handelt es sich hier um boolesche Bedingungen, die im Werkzeug geprüft und miteinander verknüpft werden müssen. Weiterhin muß auf die Änderung des Zustands reagiert werden: Wird ein Listeneintrag selektiert oder werden Schreibrechte an einem Objekt gewährt, muß das GUI an die neue Situation angepaßt werden.

genform enthält Komponenten, mit denen Bedingungen überwacht und auf Änderungen reagiert werden kann. Abbildung 3.30 zeigt einen Ausschnitt aus der Klassenhierarchie. Jede Condition hat den Zustand *true* oder *false*. Eine ComplexCondition kann mehrere Condition-Objekte enthalten und bestimmt ihren Zustand aus den Zuständen dieser Objekte. Ändert sich der Zustand einer enthaltenen Condition, wird die ComplexCondition informiert und bestimmt ihren Zustand neu. Sie meldet sich dazu als ChangeListener bei den enthaltenen Condition-Objekten an.

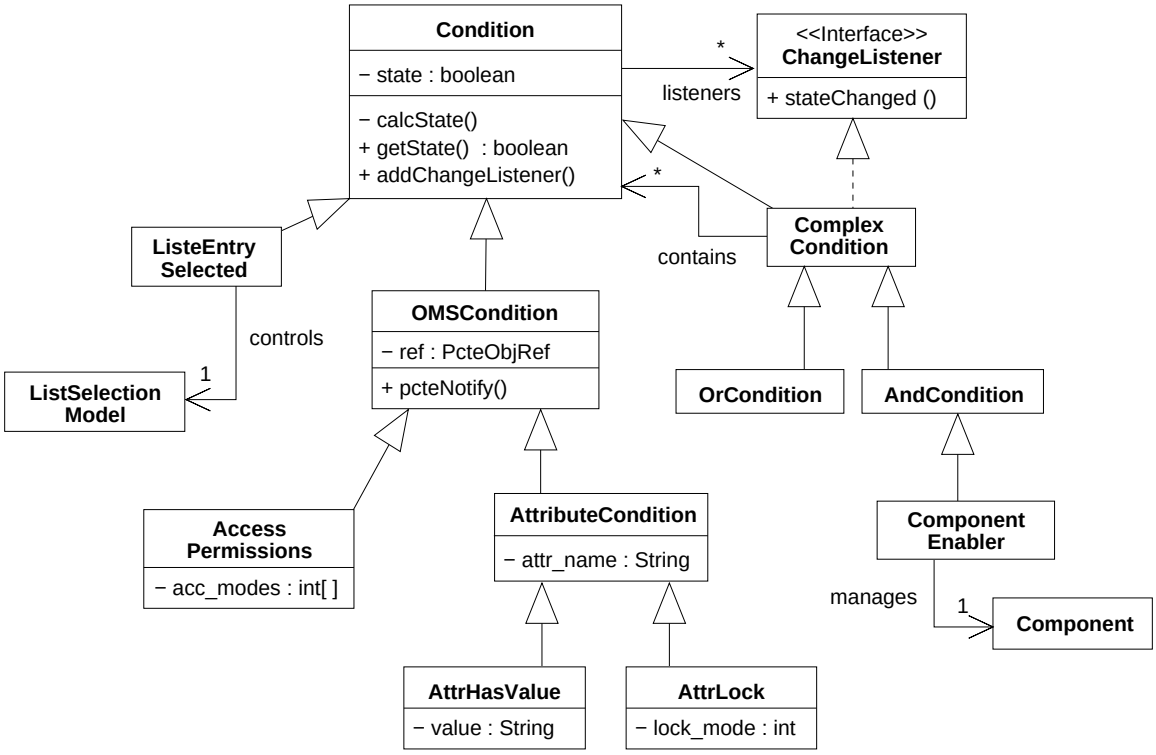


Abbildung 3.30: Ausschnitt aus der Klassenstruktur für Bedingungen

ListEntrySelected überwacht die Selektierung von Listeneinträgen: Ändert sich die Selektierung, wird ein *GUI event* ausgelöst. Die ListEntrySelected-Bedingung bestimmt daraufhin ihren Zustand neu und informiert bei einer Änderung eventuelle ChangeListener-Objekte. OMSCondition-Objekte überwachen den Zustand der Objektbank. Sie werden per Benachrichtigungsmechanismus über Änderungen informiert.

Die Klasse ComponentEnabler ist ein Beispiel für eine Condition, die den GUI-Zustand direkt beeinflußt: Sie enthält beliebige Condition-Objekte und verknüpft deren Zustände mit einem

logischen Und. Ist der resultierende Zustand *true*, wird die zugeordnete GUI-Komponente aktiviert, andernfalls deaktiviert. Um die Aktivierung eines Eingabefelds zu steuern, müssen also nur die passenden Condition-Objekte für Zugriffsrechte und Sperren erzeugt, in einen ComponentEnabler eingefügt und dieser dem Eingabefeld zugeordnet werden.

OMS-Komponenten

OMS-Komponenten fassen Funktionen zusammen, die an vielen Stellen im Werkzeug für den Zugriff auf das OMS benötigen werden:

- Lesen und Schreiben von Attributwerten, Erzeugen und Löschen von Objekten und Links
- Pufferung von Daten zur Erhöhung der Zugriffsgeschwindigkeit
- Überwachen von Änderungen an Ressourcen
- Prüfen und Ändern der Rechtesituation an Objekten
- Prüfen der Sperrsituation und Anfordern von Sperren
- Prüfen von Typrechten
- Überwachen von Änderungen der Rechte- und Sperrsituation

Durch die Zusammenfassung der Funktionen wird der Implementierungsaufwand reduziert und Redundanz vermieden; außerdem wird sichergestellt, daß sich die verschiedenen Werkzeuge in ähnlichen Situationen gleich verhalten, etwa bei der Berücksichtigung von Sperren.

Aufgaben, die in vielen GUI anfallen, sind die Zwischenspeicherung, Anzeige, Manipulation und Überwachung von Attributwerten und Beziehungen. Am Beispiel eines Puffers für ausgehende Links eines Objekts soll die Funktion der OMS-Komponenten verdeutlicht werden.

Linkpuffer. GUI für Listen, Tabellen, Baum- und Diagrammdarstellungen müssen die ausgehenden Links eines Wurzelobjekts verwalten. Die Links können Kompositionsbeziehungen oder Assoziationen repräsentieren. Das GUI benötigt folgende Funktionen:

- Einlesen und Zwischenspeichern der benötigten Links. Meist wird nur eine Teilmenge der ausgehenden Links eines Objekts in einem GUI angezeigt. Mögliche Auswahlkriterien sind der Typ des Links, der Typ des Zielobjekts oder weitere Eigenschaften, wie Attributwerte des Links oder Eigenschaften des Zielobjekts. Ein Werkzeug kann wiederum mehrere GUI mit verschiedenen Linkmengen enthalten. Um die Zugriffsgeschwindigkeit zu erhöhen, sollten alle Links mit einer OMS-Operation eingelesen werden.
- Erzeugen von Links und Objekten. In beiden Fällen müssen die Schlüsselattribute des Links passend gewählt werden; beim Erzeugen von Objekten müssen die Zugriffsrechte gesetzt werden.
- Überwachen von Änderungen der Linkmenge. Beim Erzeugen und Löschen von Links muß die verwaltete Linkmenge angepaßt und das betroffene GUI aktualisiert werden.

In *genform* übernimmt die LinkCache-Komponente diese Aufgaben. In Abbildung 3.31 ist skizziert, wie der LinkCache genutzt wird: Ein LinkCache verwaltet die ausgehenden Beziehungen von genau einem Objekt. Er enthält Paare aus Linkname und einer Referenz auf

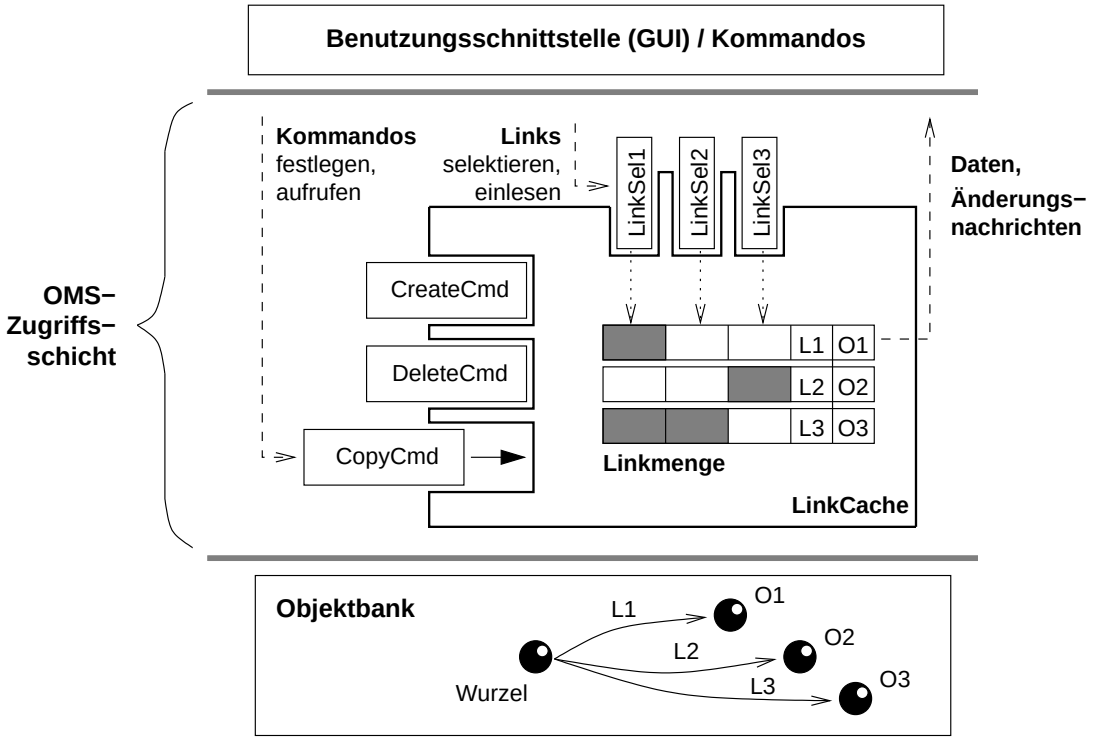


Abbildung 3.31: Verwendung der LinkCache-Komponente

das Zielobjekt. Beim Einrichten müssen zunächst die auf dem Puffer ausführbaren Kommandos erzeugt und im Puffer installiert werden. In den Kommandos ist implementiert, wie Schlüsselattribute vergeben, Zugriffsrechte gesetzt und erzeugte Ressourcen initialisiert werden. Diese Implementierung kann zum einen technische Eigenschaften des Schemas berücksichtigen, zum anderen auch Besonderheiten der Werkzeugkomponenten, die auf dem Puffer arbeiten. Somit ist der LinkCache leicht um werkzeugspezifische Kommandos erweiterbar. Die LinkCache-Kommandos werden dann von den Werkzeugkommandos aufgerufen.

Im nächsten Schritt werden die Eigenschaften der zu verwaltenden Links und Objekte mit *Linkselektoren* beschrieben. Diese Eigenschaften können sich auf die Typen von Links und Zielobjekten oder auch auf Attributwerte beziehen. Nach der Definition der Linkselektoren wird der Puffer geladen. Da alle Links eines Werkzeugs in einem Puffer verwaltet werden, ist hierzu nur eine OMS-Operation nötig.

Werkzeugkomponenten greifen über den Linkselektor auf die Daten im Puffer zu. Der Puffer überwacht die Änderungen am Objekt und wird automatisch aktualisiert. Änderungsnachrichten werden an Werkzeugkomponenten weitergeleitet, wenn sie zum vorher definierten Linkselektor passen.

Der LinkCache übernimmt auch die Prüfung von Zugriffsrechten und Sperren. Eine Werkzeugkomponente kann damit ermitteln, ob eine bestimmte Operation auf dem Wurzelobjekt ausführbar ist und das Kommando in der Benutzungsschnittstelle passend aktivieren. Der Zustand, ob ein Kommando ausführbar ist, wird in einer Condition gekapselt. Damit wird auch gewährleistet, daß die Änderung des Zustands (etwa Zurückziehen einer Sperre durch einen parallelen Prozeß) zu einer Änderung im GUI (Aktivieren des Löschkommandos) führt.

In Abbildung 3.32 sind die Condition-Objekte zur Aktivierung des Löschkommandos dargestellt. Wichtig ist hier die Verknüpfung von lokalen GUI-Ereignissen (List Entry Selected) mit Ereignissen, die vom verteilten Benachrichtigungsmechanismus des OMS ausgelöst werden.

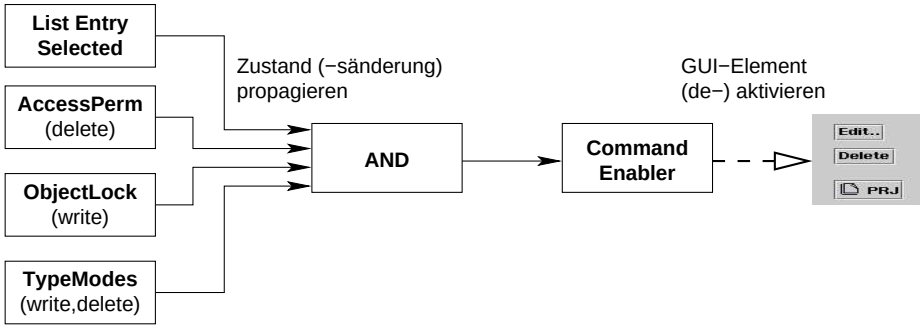


Abbildung 3.32: Aktivierung des Löschkommandos

3.3.3 Anpassung und Erweiterung

Werkzeugschemata beschreiben die Eigenschaften von Werkzeugen. Die Werkzeuge werden zur Laufzeit aus Werkzeugkomponenten zusammengesetzt und letztere mit dem Datenmodell und den Werkzeugparametern versorgt. Der Werkzeugentwickler muß also nicht wissen, wie das Framework, aus dem die Komponenten stammen, intern aufgebaut ist, und wie es funktioniert. Diese Art der Wiederverwendung wird daher auch als *black box-Wiederverwendung* [25] bezeichnet. Sie hat den Vorteil, daß Werkzeuge einfach und schnell gebaut werden können – andererseits ist die Flexibilität beim Werkzeugbau eingeschränkt, da Werkzeuge nur solche Funktionen und Darstellungen enthalten können, die bereits beim Entwurf des Frameworks vorgesehen waren.

Eine weiterführende Anpassung kann durch Änderung und Erweiterung von Framework-Komponenten erreicht werden. Diese setzt natürlich die Kenntnis des Frameworks, seiner Architektur und Funktionsweise voraus und wird daher als *white box-Wiederverwendung* [327] bezeichnet. Anpassungen und Konkretisierungen finden meist an bestimmten Stellen im Framework statt, den *Variationspunkten* oder *hot spots* [135].

Folgende Komponenten müssen häufig beim Bau von CASE-Werkzeugen angepaßt werden:

- Komponenten, die die graphische oder textuelle Darstellung von Dokumenteinträgen und Beziehungen implementieren.
- Komponenten, die Werkzeugkommandos implementieren, etwa zur Konsistenzsicherung oder Transformation von Dokumenten.

Ergebnis der Anpassung ist eine *werkzeugspezifische Komponente*, die in einem oder einer geringen Zahl von Werkzeugen eingesetzt werden kann. Oft zeigt sich, daß eine zunächst werkzeugspezifisch entwickelte Komponente hinreichend allgemein ist, daß sie in das Framework aufgenommen werden kann. Wird die Komponente so gestaltet, daß sie mittels Werkzeugschemata parametrisiert werden kann, ist eine spätere *black box*-Wiederverwendung möglich,

und das Framework wächst und reift parallel zur Werkzeugentwicklung [135]. Pree [278] bezeichnet daher *black box*-Frameworks auch als "Endstadien" einer *white box*-Entwicklung.

Wichtig ist, daß werkzeugspezifische Komponenten ohne zusätzlichen Programmieraufwand in das Werkzeug integriert werden können: Der Typ der gewünschten Komponente wird in einem Werkzeugparameter angegeben und von der 'umgebenden' Werkzeugkomponente gelesen. Das folgende Beispiel verdeutlicht die Wiederverwendung und Anpassung von Werkzeugkomponenten.

Beispiel 3.5 (Graphische Darstellung von Dokumenten) Abbildung 3.33 zeigt einen Ausschnitt aus der Klassenhierarchie für die graphische Darstellung von Dokumenten. Im unteren Teil der Abbildung sind die von den Klassen implementierten Darstellungen skizziert. Element ist die Basisklasse für Dokumenteinträge. Sie enthält eine Objektreferenz auf das Objekt, das den Eintrag in der Objektbank repräsentiert, und stellt den Dokumenteintrag als Rechteck dar. Im fertigen Werkzeug muß die gewünschte Darstellung noch in der Operation paint implementiert werden.

Die abgeleitete Klasse AttributedElem verwaltet eine Menge von Attributen des Eintrags. Die Attribute können im Element dargestellt und vom Benutzer in Eingabefeldern direkt editiert werden. Außerdem wird die Darstellung automatisch aktualisiert, wenn sich ein Attributwert in der Objektbank ändert. OMS-Komponenten übernehmen dabei die Überwachung von Änderungen und das Prüfen von Zugriffsrechten und Sperren. AttributedElem implementiert aber keine Darstellung für das Element.

In der Vererbungshierarchie implementiert erstmals NamedElement eine konkrete graphische Darstellung, wie sie in verschiedenen Dokumenttypen benötigt wird: Das Element wird als Rechteck dargestellt mit einem Attributwert in der Mitte. Eine *black box*-Wiederverwendung der Komponente ist möglich, weil der Name des anzuzeigenden Attributs über einen Werkzeugparameter spezifiziert und damit an das konkrete Werkzeug angepaßt werden kann. Um einen Entitätstyp im ER-Editor graphisch darzustellen (vgl. Abbildung 3.7 auf Seite 94), sind also zwei Werkzeugparameter nötig. Sie werden dem Objekttyp zugeordnet, der Entitätstypen im Werkzeugschema des graphischen Editors repräsentiert:

```
// Darstellungskomponente festlegen
graph.class_name = "genform.graph.elem.NamedElement"
// Name des anzuzeigenden Attributs
graph.attribute_name = "entity_type_name"
```

Die Komponente RelType implementiert die Darstellung eines Beziehungstyps im ER-Diagramm. Sie überschreibt lediglich die Darstellungsoperation paint() von NamedElement und zeichnet statt des Rechtecks eine Raute.

Ein Beispiel für eine *white box*-Wiederverwendung ist die Komponente CollElement. Sie implementiert die graphische Darstellung eines Objekts in einem Kollaborationsdiagramm. Da in der Darstellung mehrere Attributwerte erscheinen, wird AttributedElem direkt erweitert und die Operationen zur Initialisierung des Elements, zum Laden der Attributwerte, zum Darstellen des Elements und zur Größenberechnung überschrieben.

□

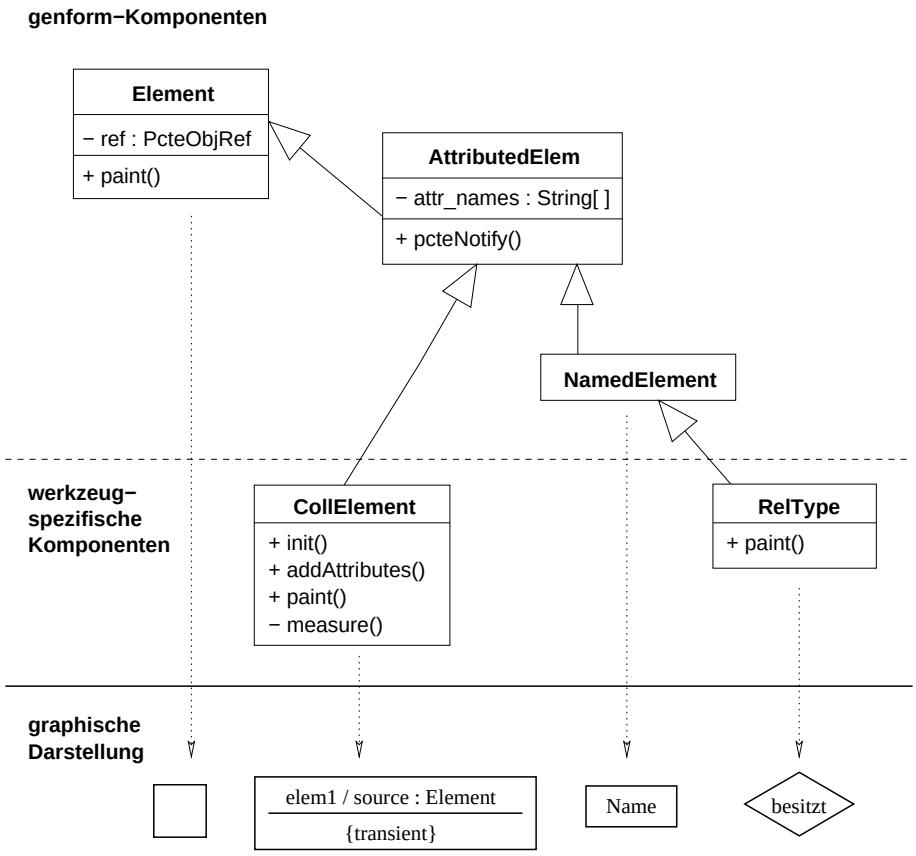


Abbildung 3.33: Ausschnitt aus der Klassenstruktur für die graphische Darstellung von Dokumenten

3.4 Beispiel: OOA-Editor

In diesem Abschnitt wird anhand eines vereinfachten Beispiels die Funktionsweise der generischen Werkzeuge erläutert. Es handelt sich um einen graphischen Editor für OOA-Klassendiagramme. Das gezeigte Werkzeug ist Teil der CASE-Werkzeugsammlung *PI-SET* [246], die graphische und textuelle Werkzeuge zur Bearbeitung verschiedener Analyse- und Entwurfsdiagramme enthält.

Zusammensetzung des Werkzeugs

In Abbildung 3.34 ist oben links der graphische Editor für Klassendiagramme dargestellt. Er enthält eine Zeichenfläche, auf der Elemente und Beziehungen eines Klassendiagramms plaziert werden können. Am linken Fensterrand befindet sich eine Werkzeugleiste, die Kommandos zum Editieren und Löschen von Einträgen enthält, sowie Knöpfe zum Erzeugen neuer Einträge.

In der Abbildung sind die Werkzeugkomponenten angegeben, die im Werkzeug verwendet werden:

– RootedFrame implementiert einen Fensterrahmen, der einem Wurzelobjekt (hier dem OOA-Diagramm) zugeordnet ist. Der Name des Objekts (hier 'elements') wird unterhalb der

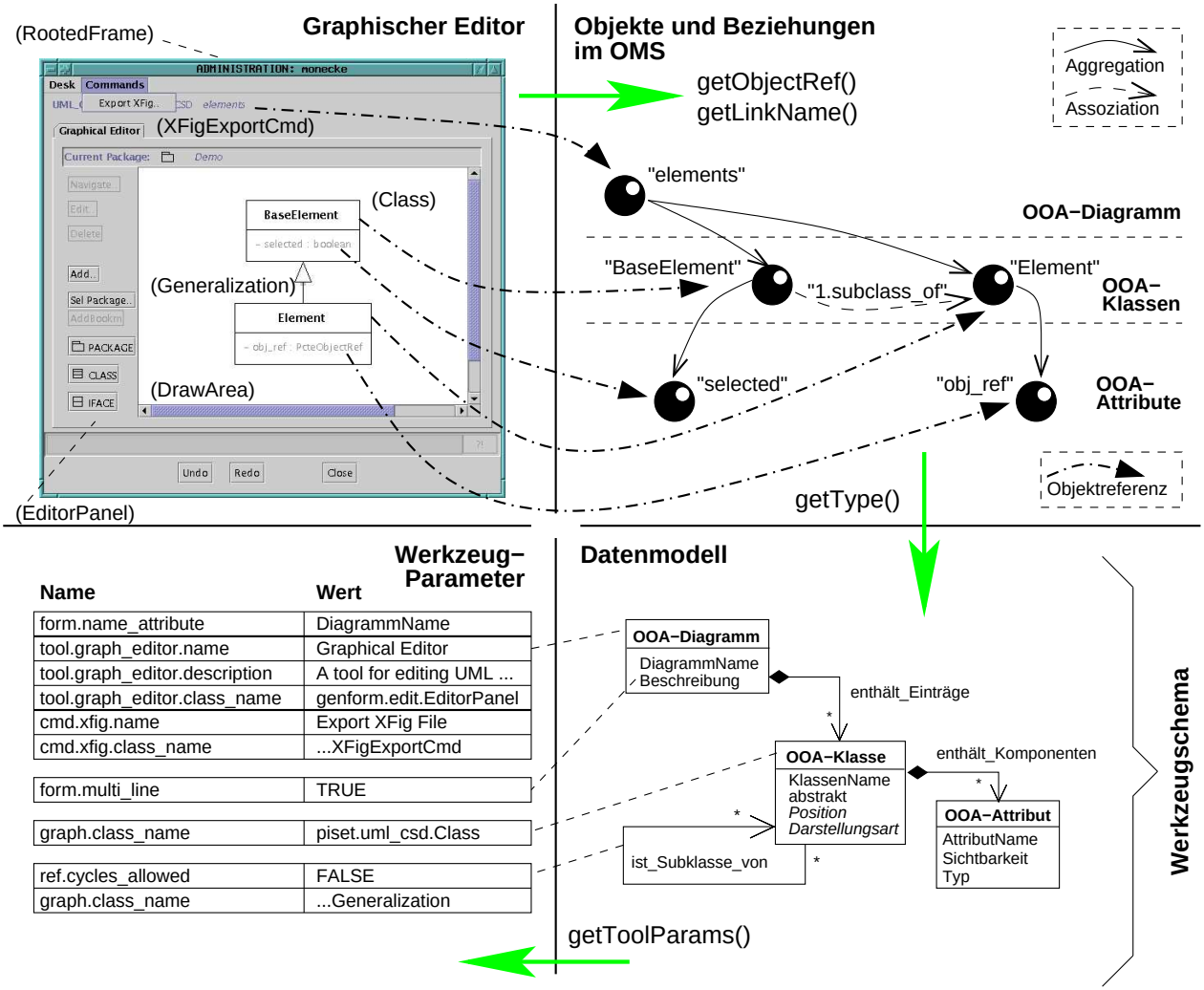


Abbildung 3.34: Funktionsweise der generischen Werkzeuge

Menüzeile angezeigt.

- Das EditorPanel wird auf der Tabulatorkarte mit der Bezeichnung 'Graphical Editor' dargestellt. Es enthält die Werkzeugleiste am linken Rand, die Anzeige des aktuellen Pakets oberhalb der Zeichenfläche und die Zeichenfläche selbst.
- Die Zeichenfläche wird durch die DrawArea-Komponente implementiert. Sie ermöglicht die Anzeige und Manipulation netzartiger Dokumente.
- Innerhalb der Zeichenfläche wird das Dokument dargestellt. Die graphischen Darstellungen von Dokumenteinträgen (hier OOA-Klassen) und Beziehungen (hier eine Vererbungsbeziehung) werden von den Komponenten Class bzw. Generalization implementiert.
- Kommandos, die auf dem Dokument ausgeführt werden können, werden im 'Commands'-Menü angeboten. Das Kommando zum Export von Graphik-Dateien im *xfig*-Format wird durch die Komponente XFigExportCmd implementiert.

Ein Werkzeug besteht also aus einer Hierarchie von Werkzeugkomponenten (vgl. Abbildung 3.35). Diese stammen aus dem *genform*-Framework (DrawArea, XFigExportCmd), oder sind

werkzeugspezifisch (Class und Generalization). Werkzeugkomponenten, die auf Objekten und

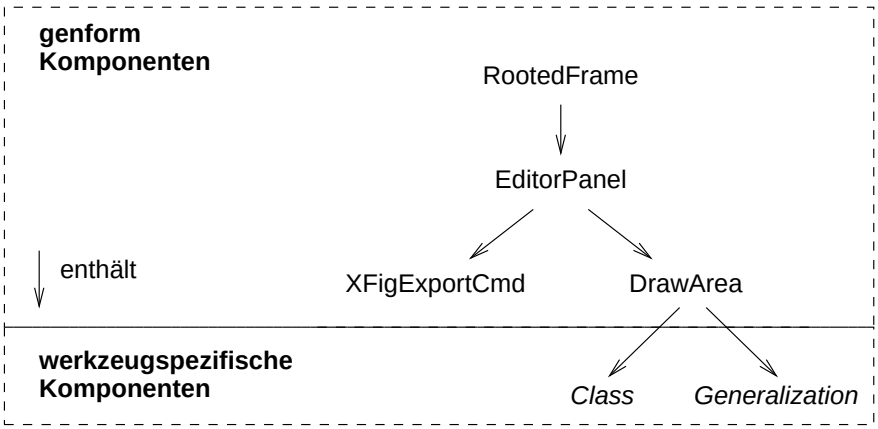


Abbildung 3.35: Hierarchie von Werkzeugkomponenten

Beziehungen im OMS operieren, enthalten Verweise auf diese Objekte (Objektreferenzen). Objektreferenzen sind in der Abbildung als strichpunktierte Pfeile dargestellt. Links werden durch ihren Namen identifiziert (1.subclass_of). Der Linkname ist bezogen auf das Objekt, von dem der Link ausgeht, eindeutig.

OMS

Das Dokument wird rekursiv durch Objekte und Beziehungen im OMS gespeichert. Die sich für das Beispieldokument ergebende Objektstruktur ist rechts oben in Abbildung 3.34 skizziert. Auf die Darstellung der Linkattribute wurde verzichtet; an den Objekten ist nur der Wert eines identifizierenden Attributs angegeben.

Es werden zwei Arten von Beziehungen unterschieden:

- *Kompositionsbeziehungen* zwischen einem Objekt und seinen Komponenten sind als durchgezogene Pfeile dargestellt (z.B. zwischen dem Diagramm und den enthaltenen Klassen).
- *Assoziationsbeziehungen* stellen lediglich einen Verweise auf ein anderes Objekt dar (z.B. von der Sub- zur Superklasse). Sie sind in der Abbildung gestrichelt dargestellt.

Das OMS bietet eine Schnittstelle an, über die Werkzeugkomponenten den Typ eines Objekts ermitteln können (*getType()*). Mit dem Typnamen können wiederum die Typeigenschaften aus der Meta-Datenbank ausgelesen werden.

Datenmodell

Das zugehörige (vereinfachte) Werkzeugschema ist im unteren Teil von Abbildung 3.34 dargestellt. Es enthält zum einen das Datenmodell des Werkzeugs (rechts); zum anderen Werkzeugparameter, mit denen die Werkzeugkomponenten parametrisiert werden (links).

Im graphischen Editor werden folgende Informationen aus dem Datenmodell ausgewertet:

- Wegen der Beziehung enthält_Einträge wird ein Kommando zum Erzeugen von OOA-Klassen angeboten. Im graphischen Editor kann das Kommando über einen Knopf in der Werkzeugleiste oder einen Eintrag im Popup-Menü der Zeichenfläche ausgelöst werden.

- Wegen der Beziehung enthält_Komponenten bieten OOA-Klassen ein Kommando zum Erzeugen von OOA-Attributen an. Das Kommando wird über einen Eintrag im Popup-Menü der Klasse ausgelöst oder im Menü des separaten Werkzeugs zur Bearbeitung der Klasse.
- Wegen der Beziehung ist_subklasse_von bieten OOA-Klassen ein Kommando zum Erzeugen von Vererbungsbeziehungen an. Das Kommando kann über einen Eintrag im Popup-Menü der Klasse oder durch Anklicken einer Klasse bei gleichzeitigem Drücken der 'Ctrl'-Taste ausgelöst werden. Im Datenmodell kann nicht festgelegt werden, ob Ausgangs- und Zielobjekt einer Beziehungen identisch sein dürfen. Dies wäre beim Erzeugen von Assoziationen im OOA-Diagramm erlaubt; bei Vererbungsbeziehungen hingegen nicht. Die Eigenschaft muß daher in einem Werkzeugparameter festgelegt und dem Beziehungstyp zugeordnet werden.
- Informationen über die Attribute der Objekttypen werden im graphischen Editor nicht ausgewertet, da in der graphischen Darstellung festgelegt wird, auf welche Attribute zugegriffen werden soll. Ein generisches Formular, das alle Attribute eines Objekts oder Links darstellt, würde hingegen die Menge der Attribute aus der Meta-Datenbank ermitteln.

Werkzeugparameter

Unten links ist in Abbildung 3.34 eine Teilmenge der Werkzeugparameter dargestellt, die Attributen, Objekt- und Linktypen zugeordnet werden. Werkzeugkomponenten können die Werkzeugparameter über die Schnittstelle *getToolParams()* auslesen.

Für Werkzeugparameter wird ein hierarchischer Namensraum verwendet: Die Werkzeugparameter, die ein komplexer Werkzeugparameter enthält, erhalten Namen mit dem gleichen Präfix. Werkzeugparameter mit dem Präfix form sind für die textuelle Darstellung relevant. Der Parameter form.name_attribute kennzeichnet das Attribut eines Objekts, dessen Wert in der Kopfzeile eines Fensters oder in Listeneinträgen zur Identifikation des Objekts angezeigt wird. Beim Objekttyp OOA-Diagramm ist dies der Diagrammname.

tool-Parameter beziehen sich auf Werkzeuge, die einem Objekttyp (oder einem Linktyp) zugeordnet werden. Mit tool.graph_editor werden die Parameter für den graphischen Editor bezeichnet: tool.graph_editor.name enthält die Bezeichnung des Werkzeugs, die im Menüeintrag zum Starten des Werkzeugs und auf der Tabulatorkarte des Werkzeugs angezeigt wird. description enthält eine Beschreibung oder einen Hilfetext zum Werkzeug. class_name enthält den Namen der Werkzeugkomponente, die das Werkzeug implementiert. Beim graphischen Editor ist dies die (generische) Komponente EditorPanel. An dieser Stelle könnte auch eine werkzeugspezifische Komponente spezifiziert werden.

Die cmd-Parameter beschreiben Kommandos, die auf einem Objekt ausführbar sind. cmd.xfig kennzeichnet das Kommando zum Exportieren von Grafikdateien im *xfig*-Format. Das Kommando wird beschrieben durch einen Namen (name) für den Menüeintrag und die Werkzeugkomponente, die das Kommando implementiert (class_name). Die verwendete Komponente genform.graph.XFigExportCmd ist generisch und stützt sich für die Darstellung auf Operationen der Elemente und Beziehungen ab.

Dem Attribut Beschreibung des Objekttyps OOA-Diagramm ist der Parameter form.multi_line zugeordnet. Er legt fest, daß im Formular ein mehrzeiliges Eingabefeld für das Attribut

angeboten wird. Die Komponente, die die graphische Darstellung von OOA-Klassen implementiert, wird durch den Werkzeugparameter graph.class_name des Objekttyps OOA-Klasse festgelegt.

Werkzeugparameter, die die Eigenschaften von Beziehungen im Dokument beschreiben, beginnen mit dem Präfix ref. Mit ref.cycles_allowed = FALSE wird für die Vererbungsbeziehung festgelegt, daß Ausgangs- und Zielobjekt nicht identisch sein dürfen. Mit graph.class_name wird, analog zum Klassensymbol, die Komponente zur graphischen Darstellung von Vererbungsbeziehungen bestimmt.

3.5 Andere Ansätze

Auch in *VSF* [272] sorgt ein zentrales Repository (*"at the heart of the environment"*) für die Datenintegration. Zusätzlich können Daten in Fremdformaten im- und exportiert werden. Die Konzepte einer Methode werden durch semantische Datenmodelle beschrieben. Den Konzepten im Modell werden Objekte für die textuelle oder graphische Darstellung zugeordnet. Die Typen dieser Objekte bestimmen die Eigenschaften der Darstellung und die verfügbaren Operationen. Unterschiedliche Sichten auf das Datenmodell werden nicht unterstützt.

HyperCASE [67] kombiniert CASE-Werkzeuge mit Hypertext-Funktionen: Der Datenbestand einer CASE-Umgebung wird als Sammlung von Dokumenten mit dokumentübergreifenden Beziehungen betrachtet. Solche Beziehungen können sich über den gesamten Lebenszyklus eines Softwaresystems erstrecken, so daß etwa die Auswirkungen einer Anforderung im resultierenden Quelltext nachvollzogen werden können. Allerdings werden keine Sichten verwendet, sondern Änderungen an überlappenden Daten von einer Komponente namens *Event Manager* propagiert. Sie ist auch für die Aktualisierung der Darstellung in Werkzeugen zuständig.

Das Metamodell von *Metaview* [113] basiert auf dem ER-Modell. Hier wird einem Objekttyp eine graphische Darstellung zugeordnet, zusammengesetzt aus graphischen Primitiven [112]. Die Eigenschaften der Objekttypen dienen als Vorgabewerte für die Instanzen und können überschrieben werden.

In *genform* kann in verschiedenen Sichten sowohl das Datenmodell als auch die Darstellung variiert werden. Zusätzlich werden auch die im Werkzeug verfügbaren Kommandos und die Zusammensetzung der Werkzeuge aus Komponenten im Werkzeugschema festgelegt.

Im *Viewer*-Prototyp [263] wird das Konzept der *Viewpoints* mit dem Ziel erprobt, eine feingranulare Prozeßunterstützung [223] zu realisieren. *Viewpoints* kapseln die verschiedenen Sichten der Beteiligten auf die Produkte und den Software-Entwicklungsprozeß – solche Sichten werden in *genform* über Werkzeugschemata realisiert.

Gille beschreibt in [129], wie graphische Editoren aus OOA-Modellen generiert werden können. Er erweitert damit den *JANUS*-Ansatz [18] auf graphische Werkzeuge. Die Darstellung von Elementen und Beziehungen wird mit einem Symboleditor erstellt; die Komponenten der generierten Werkzeuge befinden sich in einer Bibliothek. Der Generator bestimmt, wie die Werkzeuge aus diesen Komponenten zusammengesetzt werden. Die Erweiterung des Ansatzes ist aufwendig: Soll in einer Anwendung ein neues primitives Symbol verwendet werden, muß zunächst die Bibliothek um neue Klassen ergänzt und der Symboleditor erweitert wer-

den. Gille schätzt den Aufwand auf "einige Tage".

In *genform* werden Werkzeuge auch aus generischen Komponenten zusammengesetzt. Die Regeln, wie die Werkzeuge zusammengesetzt sind und welche Eigenschaften sie haben, sind dabei nicht in einem Generator implementiert, sondern in den Werkzeugkomponenten selbst, die das Werkzeugschema (Datenmodell und Werkzeugparameter) interpretieren. Der Werkzeugentwickler kann die Komponenten leicht erweitern und in Werkzeuge einbinden.

Die Idee, das Datenmodell des Werkzeugs zu interpretieren, ist aus *ToolFrame* [69] übernommen. *ToolFrame* ist eine erweiterbare *Upper-CASE*-Umgebung, also sowohl eine ausführbare Umgebung (wie *PI-SET*), als auch ein Framework für CASE-Werkzeuge (wie *genform*). In *ToolFrame* wird ebenfalls H-PCTE als Repository verwendet, allerdings werden Zugriffskontrollen, der verteilte Benachrichtigungsmechanismus, Transaktionen und Sperren nicht genutzt, da sie zum Teil in der damaligen Implementierung nicht zur Verfügung standen.

Allerdings ist *ToolFrame* schwieriger zu erweitern als Umgebungen, die mit *genform* gebaut wurden:

1. Es steht kein Mechanismus zur Verfügung, über den werkzeugspezifische Komponenten integriert werden könnten. Die kleinste Einheit einer Umgebung, die manipuliert werden kann, ist das Werkzeug. Um also einen graphischen Editor zu integrieren, der ein spezielles Export-Kommando anbietet, muß eine neue Werkzeugklasse (GraphischerEditorMitExport-Kommando) implementiert werden, die das Kommando enthält.

   In *genform* bieten die generischen Komponenten 'Steckplätze' an, in die werkzeugspezifische Komponenten einrasten können. Im obigen Beispiel würde das Export-Kommando durch eine werkzeugspezifische Komponenten implementiert und über einen Werkzeugparameter in das Werkzeug eingebaut.

2. Generische Komponenten und werkzeugspezifische werden nicht getrennt verwaltet. Alle Komponenten sind Teil des Frameworks, auch wenn eine Komponente nur für ein Werkzeug nutzbar ist. In *genform* werden beide Arten von Komponenten getrennt und werkzeugspezifische Komponenten beim Werkzeug verwaltet. Erweist sich eine werkzeugspezifische Komponente als hinreichend allgemein, um auch in anderen Werkzeugen verwendbar zu sein, kann sie in das Framework integriert werden. Auch werkzeugspezifische Komponenten können mit Werkzeugparametern beeinflußt werden und sind daher flexibel nutzbar.

3. In *ToolFrame* gibt es drei Arten von generischen Werkzeugen: Einen Matrix-Editor, einen graphischen Editor und einen Analysator. Werkzeuge werden den Dokumenttypen in einer *Werkzeugdatenbank* [68] zugeordnet. Die graphische Darstellung von Elementen und Beziehungen wird aus graphischen Primitiven zusammengesetzt und in einer *Layout-Datenbank* [245] gespeichert. In der Layout-Datenbank kann nur eine Darstellung pro Typ gespeichert werden; verschiedene Darstellungen sind durch Umbenennen von Typen in verschiedenen SDS möglich. Die Beeinflussung der Darstellung (Ein- und Ausblenden von Attributen und Operationen im Klassensymbol) ist nicht vorgesehen. Die Einträge für die Layout- und Werkzeugdatenbank werden im Quelltext erzeugt. Eine Änderung ist daher nur mit einigen Vorkenntnissen möglich und erfordert das Neuübersetzen der Umgebung. Auch kann an dieser Stelle die Konsistenz zwischen den Einträgen und dem Datenbankschema nicht geprüft werden.

   In *genform* sind die Beschreibungen den Typen im Datenbankschema zugeordnet (Para-

meter im Werkzeugschema): Es muß also nur das Werkzeugschema geändert werden, um Werkzeuge zu ändern oder zu erweitern. In verschiedenen Varianten eines Werkzeugschemas können die Werkzeugeigenschaften variiert, gleichzeitig Gemeinsamkeiten wiederverwendet werden. Werden Werkzeugkomponenten geändert, müssen nur diese neu übersetzt werden.

4. Die Framework-Klassen von *ToolFrame* sind stark auf die vorhandenen Werkzeuge spezialisiert. Die Erweiterung von *ToolFrame* um neue Arten von Werkzeugen ist daher aufwendig, da zu viele Annahmen über die Funktion der Werkzeuge in den Komponenten enthalten sind. Allerdings war die Erweiterung des Frameworks durch den Werkzeugentwickler auch nicht vorgesehen.

   In *genform* kann der Werkzeugentwickler Framework-Komponenten erweitern. Betroffen ist meist nur eine geringe Anzahl von Komponenten, die Kommandos oder die graphische Darstellung von Dokumenten implementieren. Beim Bau neuer Werkzeugtypen kann der Werkzeugentwickler Komponenten für den Datenbankzugriff oder die Benutzungsschnittstelle verwenden, was den Implementierungsaufwand reduziert.

MetaEdit+ [197] ist ein kommerzielles Meta-CASE-System. Dokumente werden mit einem erweiterten ER-Modell (GOPRR) beschrieben. Es gibt Tabellen-, Matrix- und graphische Darstellungen. Letztere werden mit einem Symboleditor spezifiziert. Dokumente können auf Konsistenz geprüft und Reports generiert werden. Da es sich um ein Smalltalk-System handelt, kann das Framework erweitert werden. Dieser Weg ist aber für den Werkzeugentwickler nur schwer gangbar, da er eine umfassende Kenntnis des Frameworks voraussetzt.

In *genform* ist die Erweiterung von Werkzeugkomponenten hingegen vorgesehen. Für einfache Änderungen wie die Implementierung einer graphischen Darstellung oder eines Kommandos muß der Werkzeugentwickler nur wenige Klassen kennen; erst umfangreiche Änderungen erfordern einen größeren, aber weiterhin überschaubaren Aufwand. Dies hat die Implementierung einiger Werkzeuge gezeigt [248, 289, 320].

Auch in anderen Ansätzen kann der Werkzeugentwickler Framework-Klassen erweitern: *MetaMOOSE* [110] ist ein *Itcl*-Framework (objektorientierte Variante von *Tcl*). Ferguson et al. legen besonderen Wert darauf, daß auch Funktionen, die im Framework nicht vorgesehen waren, leicht in die Werkzeuge eingebaut werden können, ohne daß Editoren oder Generatoren aufwendig geändert werden müssen. Auch arbeitet der Werkzeugentwickler auf einer hohen Abstraktionsebene, da er die Konzepte des Frameworks wiederverwenden und erweitern kann. Können nur einzelne Routinen oder Makros programmiert werden (wie in *ToolBuilder* [5] mit *Easel* oder in *KOGGE* [93] mit einer Sprache, die Modula-2 ähnelt), muß der Werkzeugentwickler auf niedriger Abstraktionsebene und mit sehr eingeschränkten Möglichkeiten zur Wiederverwendung arbeiten.

In *MetaMOOSE* wird das Datenmodell nicht separat definiert, sondern durch spezialisierte Framework-Klassen beschrieben. Sichten können durch Vererbungs- oder Delegationsbeziehungen realisiert werden – müssen also, anders als in *genform*, programmiert werden.

Serendipity [143], eine Editierumgebung mit Prozeßunterstützung, basiert auf dem *MViews*-Framework [147]. Im Framework sind alle benötigten Dienste, etwa zur Datenverwaltung und zum Austausch von Ereignissen, implementiert. In [150] beschreiben Grundy und Venable, wie sie die Meta-Modellierungssprache *CoCoA* zur Spezifikation von Werkzeugen nutzen. Im Vordergrund steht die Integration verschiedener Methoden wie ER und NIAM [254]: Für

beide Methoden wird je ein spezielles Datenmodell definiert sowie eines, das gemeinsame Konzepte enthält. Um die Daten zu aktualisieren, wird der Benachrichtigungsmechanismus des Frameworks genutzt. Die Autoren schätzen den Anteil des Quelltextes, der für die Definition der Datenmodelle und Sichten benötigt wird, auf über 60 % und planen daher, den Quelltext direkt aus dem CoCoA-Modell zu generieren. Auch im *MViews*-Nachfolger *JViews* [145] werden Datenmodelle und Sichten programmiert. *JViews* verwendet *Java Beans*. Der Bau von Umgebungen wird hier durch Meta-Werkzeuge unterstützt [144].

Die *genform*-Komponenten interpretieren Werkzeugschemata, also Datenmodelle und Werkzeugparameter, so daß kein Quelltext zur Implementierung von Datenmodellen und Sichten geschrieben werden muß. Datenmodelle werden mit Meta-Werkzeugen graphisch spezifiziert. Mehr zu den Meta-Werkzeugen in Abschnitt 5.4.

genform nutzt zahlreiche Dienste des OMS, was den Implementierungsaufwand für das Framework verringert und die Wartung und Einarbeitung erleichtert. Die Zusammensetzung von Werkzeugen aus Komponenten ist in *genform* allerdings weniger flexibel als in *JViews*, da in *genform* nur bestimmte Typen von Komponenten an bestimmten Stellen im Werkzeug eingesetzt werden können – in *JViews* müssen sie lediglich den *Java Beans*-Konventionen entsprechen. Allerdings ist es hier auch schwieriger, eine Komponente in ein Werkzeug einzupassen, da sie die benötigten Schnittstellen aufweisen und Ereignisse verarbeiten muß.

Auch in *JKogge* [192], dem komponentenbasierten Nachfolger von *KOGGE* [93], ist der Bauplan der Werkzeuge vorgegeben: Komponenten heißen hier *plug-ins* und rasten in *slots* des Basissystems ein. Allerdings entsprechen hier die *plug-ins* jeweils ganzen Werkzeugen (die per Internet verteilt werden können) und sind nicht generisch, sondern Dokumenttypspezifisch programmiert.

In *KOGGE* werden verschiedene Sprachen zur Beschreibung des Datenmodells, von Einschränkungen (*constraints*), der Menüstruktur, den Zuständen und Funktionen von Werkzeugen verwendet. Kritisch ist hier der hohe Lernaufwand für die verschiedenen Sprachen und die geringe Flexibilität bei unvorhergesehenen Erweiterungen, die eine Anpassung des Basissystems (*Ur-KOGGE*) erfordern würden.

IPSEN [212] kommt mit weniger Sprachen aus: Graphersetzungsregeln beschreiben das Schema von Dokumenten und mögliche Editieroperationen. Aus dem Schema wird der Quelltext für einen syntaxgesteuerten Editor generiert, der übersetzt und zusammen mit dem *IPSEN*-Framework zu einer lauffähigen Anwendung gebunden wird. Eine Erweiterung der Werkzeuge durch *handcoding* in Modula-2 oder C ist aber vorgesehen. IPSEN nutzt den Graphenspeicher *GRAS* [211]. Das Datenmodell ähnelt dem in *genform* (Knoten und Kanten); allerdings können Kanten in *IPSEN* keine Attribute besitzen.

Der *IPSEN*-Ansatz ist sehr elegant, da die Struktur der Dokumente und die Editieroperationen in einer Spezifikation zusammengefaßt sind. Allerdings sind der Generator, der die Graphersetzungsregeln in Quelltext übersetzt, und das *IPSEN*-Framework, das generische Werkzeugfunktionen enthält, sehr komplex. Es stehen keine Mechanismen zur Verfügung, um die bei der Anpassung und Erweiterung von Werkzeugen entstandenen Routinen andernorts wiederverwenden zu können.

In *genform* sind die generischen Werkzeugfunktionen in den Werkzeugkomponenten implementiert, die direkt die Werkzeugschemata interpretieren. Ein Generator, der den *glue code* zwischen den Framework-Komponenten erzeugt, ist also nicht nötig. Eine Erweiterung der

Werkzeuge ist einfach durch Spezialisierung von Framework-Komponenten möglich, wobei Vererbung, Abstraktion und Kapselung ausgenutzt werden können.

Von den vorgestellten Ansätzen berücksichtigen nur *Serendipity* und der *Viewer*-Prototyp das Vorgehensmodell der Methode, die unterstützt werden soll. In *genform* sind die Datenmodelle von Dokumenten und Prozeß integriert, so daß Beziehungen zwischen Dokument- und Prozeßdaten verwaltet werden können. Die Prozeßmaschine kann auf Benutzeraktionen reagieren und die Menge der ausführbaren Kommandos einschränken. Mehr zur Prozeßunterstützung mit *genform* im nächsten Kapitel.

Kapitel 4

Prozeßunterstützung mit *genform*

Im vorangegangenen Abschnitt wurde beschrieben, wie mit *genform* CASE-Werkzeuge gebaut werden. Die Werkzeuge unterstützen die Konzepte und Notationen der gegebenen Methode und bieten Dokumenttyp-spezifische Kommandos an. Zur vollständigen Unterstützung einer Methode muß auch ihr Vorgehens- oder Prozeßmodell berücksichtigt werden. Es kann zum einen auf Papier in einem Methoden- oder Projekthandbuch notiert werden und den Entwicklern und Managern als Richtlinie für die Projektdurchführung dienen (*off-line guidance* [229]).

Die andere Möglichkeit ist, das Prozeßmodell so weit zu formalisieren, daß es von einer Prozeßmaschine ausgeführt werden kann [267]. Die Prozeßmaschine verwaltet einen virtuellen 'internen' Prozeßzustand, der durch Rückmeldungen von Entwicklern und Managern mit dem realen 'externen' Prozeßzustand abgeglichen wird. Sie leitet Entwickler und Manager dabei an, ihre Aufgaben gemäß Prozeßmodell durchzuführen (*on-line guidance*). Ein wichtiger Aspekt ist hier, daß die CASE-Werkzeuge mit dem Prozeßmodell und dem Prozeßzustand integriert sein müssen: Entwickler sollen direkt über ihre Werkzeuge mit dem Prozeß interagieren können. Die *Prozeßunterstützung* umfaßt [60]:

1. Ein *Prozeßmodell*, das den Software-Entwicklungsprozeß für alle Beteiligten hinreichend genau beschreibt, sowie Methoden, Sprachen und Werkzeuge zur Definition und Manipulation des Modells.
2. *Prozeßwerkzeuge*, mit denen die Prozeßausführung (interaktiv) gesteuert und überwacht wird. Voraussetzung für den Einsatz der Prozeßwerkzeuge ist ein existierendes Prozeßmodell.
3. Eine *Prozeßmaschine*, die das Prozeßmodell interpretiert und auf Basis des Prozeßmodells und des aktuellen Prozeßzustands den weiteren Prozeßfortschritt festlegt.
4. Die *Einsatzumgebung* und technische Infrastruktur, um die genannten Bestandteile nutzen und integrieren zu können.

Art der Prozeßunterstützung

Die Prozeßunterstützung soll dafür sorgen, daß der Prozeß gemäß Prozeßmodell ausgeführt wird, also der tatsächliche Ablauf bezogen auf das Prozeßmodell 'korrekt' ist. Natürlich muß dazu das Verhalten der Beteiligten im Prozeß beeinflußt werden.

Die *Striktheit* [20] der Prozeßunterstützung bestimmt, wie frei der Benutzer einer PSEU bei der Ausführung seiner Aufgaben ist. Unterschieden werden [89]:

- *Passive guidance*: Die PSEU schlägt auf Anfrage die nächste Aktion vor, greift aber nicht selbständig in die Arbeit des Benutzers ein. *Passive guidance* kann auch mit einem Methodenhandbuch erreicht werden; die elektronische Variante stellt einen 'Was nun?'-Knopf zur Verfügung und liefert die Antwort basierend auf dem Prozeßmodell, dem aktuellen Prozeßzustand und der Rolle des Benutzers.
- *Active guidance*: Vorschläge für das weitere Vorgehen werden nicht auf Anfrage des Benutzers geliefert, sondern automatisch abhängig vom aktuellen Prozeßzustand. Beispielsweise können neue Aufgaben in die Aufgabenliste des Benutzers eingetragen werden, der Benutzer über wichtige Ereignisse informiert oder Warnungen und Hinweise angezeigt werden.
- *Process enforcement*: Es wird nur die Art der Prozeßausführung zugelassen, die das Prozeßmodell vorschreibt. Dies kann nur erreicht werden, indem der Zugriff auf Daten und Werkzeuge eingeschränkt wird – ein Benutzer kann in einer gegebenen Situation also nur die erlaubten Aktionen ausführen. *Process enforcement* ist mit komplexen interaktiven Werkzeugen schwierig zu erreichen, da zwar das Starten des Werkzeugs kontrolliert werden kann, selten aber die Ausführung einzelner Kommandos.
- *Process automation*: Teile des Prozesses werden automatisch ausgeführt. Dies ist nur bei komplexen und stereotypen Abläufen sinnvoll, die mit Werkzeugen automatisierbar sind. Ein Beispiel ist das Werkzeug *make*, das die Abläufe beim Generieren eines Systems automatisiert.

Die Beeinflussung ihrer Arbeit durch eine Prozeßmaschine kann bei den Beteiligten extreme Reaktionen hervorrufen: Ablehnung und Rebellion auf der einen, Aufgabe der Verantwortung und blindes Befolgen der Vorgaben auf der anderen Seite [11]. Allerdings haben planorientierte Methoden bei der Entwicklung 'kritischer' Anwendungen durchaus ihren Platz [36]; somit scheinen auch Mechanismen sinnvoll, die die Einhaltung dieser Methoden unterstützen und sicherstellen [209]. Letztendlich wird nicht durch die PSEU, sondern durch das gewählte Prozeßmodell festgelegt, wie fein Aufgaben zerlegt, wie der Zugriff auf Dokumente und Werkzeuge geregelt, und welche Anforderungen an unterschiedliche Rollen gestellt werden – also wie frei sich die Beteiligten im Prozeß bewegen können.

An dieser Stelle soll es nicht um den Entwurf von Prozeßmodellen gehen und darum, in welchen Situationen welche Arten der Prozeßunterstützung angemessen sind. Statt dessen geht es um die Frage, wie ein Mechanismus zur Steuerung und Überwachung von Prozessen technisch realisiert werden kann, der möglichst viele Einflußmöglichkeiten bietet – im vorgestellten Ansatz zur Vorgabe der ausführbaren Aufgaben, zur Steuerung der Datenflüsse und zur Einschränkung der zugreifbaren Dokumente sowie zur Anpassung der Werkzeuge an die aktuelle Prozeßsituation.

Im folgenden wird beschrieben, wie die Prozeßunterstützung in PSEU realisiert wird, die auf ***genform*** basieren. Die Beschreibung folgt dabei der von Dowson und Fernström [89] vorgenommenen Unterteilung der Prozeßunterstützung in die Bereiche Prozeßmodellierung (*process definition*), Instantiierung des Prozesses (*process enactment*) und die Ausführung des Prozesses (*process performance*). Die dabei verwendeten Prozeßmodell-Beispiel sind nicht als Vorschläge für den Entwurf solcher Modelle, sondern als Beispiele zur Demonstration der technischen Möglichkeiten gedacht.

4.1 PSEU-Architektur

Die Architektur von PSEU, die mit *genform* konstruiert werden, entspricht Architektur 2 aus Abbildung 2.3 auf Seite 57: Die Prozeßdaten werden in einem zentralen Repository verwaltet, die Prozeßausführung von verteilten Prozeßmaschinen gesteuert. Jeder PSEU-Ausführung ist genau eine Prozeßmaschine zugeordnet (siehe Abbildung 4.1).

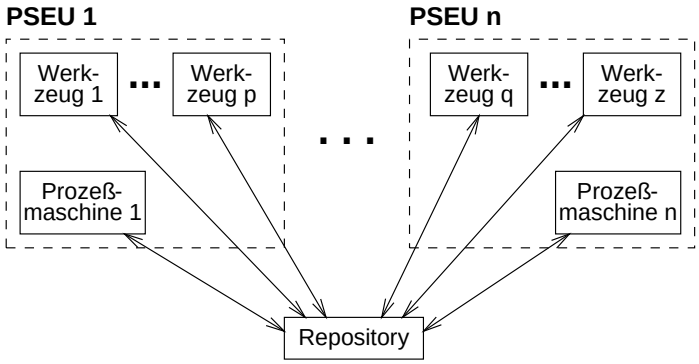


Abbildung 4.1: Architektur von PSEU auf Basis von *genform* anhand von Beispiel-Instanzen

Das Repository verwaltet

- das Prozeßmodell, das die Aufgaben des aktuellen Software-Entwicklungsprozesses beschreibt und die Beziehungen zwischen Aufgaben. Auf das Prozeßmodell greift die Prozeßmaschine während der Prozeßdurchführung zu.
- den aktuellen Prozeßzustand, der die aktuelle Projektsituation reflektiert. Er enthält die Zustände von Aufgaben (in Bearbeitung, abgeschlossen), die Zuordnung von Entwicklern und Dokumenten zu Aufgaben, Ist- und Sollaufwände, sowie Informationen über den Bearbeitungszustand, die Qualität und Korrektheit von Dokumenten.
- die Software-Dokumente, die mit CASE-Werkzeugen erzeugt, bearbeitet, analysiert und transformiert werden. Es werden keine externen Speicher für Dokumente unterstützt.

CASE-Werkzeuge, Prozeßwerkzeuge und die Prozeßmaschine arbeiten also ausschließlich auf den Daten im Repository – dadurch wird die Architektur der Umgebung einfacher und der Aufwand für die Implementierung reduziert. Das Repository sorgt für die Datenintegration von Werkzeugen und Prozeßmaschine: Um den parallelen Betrieb mehrerer PSEU mit jeweils mehreren Werkzeugen und einer eigenen Prozeßmaschine zu ermöglichen, werden folgende Dienste des Repositories genutzt[1]:

- *Zugriffskontrollen* regeln, welche Benutzer welche Änderungen mit den Werkzeugen der PSEU durchführen dürfen. Zu diesen Änderungen zählen das Ändern, Erzeugen und Löschen von Software-Dokumenten ebenso wie die Manipulation des Prozeßzustands – etwa durch das Erzeugen neuer Aufgaben oder das Ändern von Aufwänden und Terminen. H-PCTE unterstützt Zugriffskontrollen auf Basis hierarchischer Benutzergruppen.

[1] Siehe auch Beschreibung der OMS-Dienste in Abschnitt 2.3.4 auf Seite 59 und von H-PCTE in Abschnitt 2.5 auf Seite 69.

– *Sperren* regeln den parallelen Zugriff mehrerer Werkzeuge oder Prozeßmaschinen auf gleiche Daten. Die Werkzeuge innerhalb einer PSEU sollten sich dabei nicht gegenseitig behindern: Sie werden von einem Benutzer ausgeführt, der seine Aktionen selbst koordinieren kann und muß. Die Änderungen von Werkzeugen verschiedener PSEU müssen hingegen über Sperren synchronisiert werden, um Probleme wie den Verlust von Änderungen zu verhindern.

  H-PCTE besitzt einen feingranularen Sperrmechanismus [270], der eine hohe Parallelität bei der Ausführung interaktiver Werkzeuge gewährleistet und auch die Ausführung nicht-interaktiver Werkzeuge unterstützt. Sperrkonflikte zwischen Prozeßmaschinen sind selten, weil die Transaktionen der Prozeßmaschine meist kurz sind, und weil bereits das Prozeßmodell für die Strukturierung der Arbeit verschiedener Entwickler sorgt. Tritt ein Sperrkonflikt auf, kann die Prozeßmaschine die Aktion abbrechen oder (ggf. für eine vorgegebene Zeit) auf die Sperrfreigabe warten.

– Der *Benachrichtigungsmechanismus* des Repositories informiert parallel ausgeführte Werkzeuge und die Prozeßmaschine über Änderungen, die im Repository stattgefunden haben. Die Kommunikation über das Repository ist die einzige Möglichkeit zum Austausch von Informationen zwischen Werkzeugen – auf den Einsatz eines *message server* (wie *FIELD* [280]) wird also verzichtet. Dadurch wird der Realisierungsaufwand und die Komplexität der PSEU verringert. Insbesondere muß der Nachrichtenversand nicht explizit implementiert werden.

  Mit dem *verteilten Benachrichtigungsmechanismus* von H-PCTE [271] können Werkzeuge die Änderung von Ressourcen im Repository überwachen: Das Repository schickt eine Nachricht, die die Werkzeuge über die durchgeführte Änderung und den neuen Zustand der Objektbank informiert – die Kommunikation muß also nicht explizit implementiert werden; es müssen lediglich die relevanten Ressourcen überwacht und auf Änderungsnachrichten geeignet reagiert werden [208].

– Mit dem *Sichtenmechanismus* können werkzeugspezifische Sichten definiert werden. Jede Sicht enthält nur den für das Werkzeug relevanten Ausschnitt aus dem gesamten Datenmodell – jedes Werkzeug arbeitet also mit einer eigenen Sicht, ist aber gleichzeitig mit den übrigen Werkzeugen über das gemeinsame konzeptuelle Schema integriert. Mit Typrechten können die vom Werkzeug durchführbaren Änderungen sehr feingranular gesteuert werden.

4.2 Prozeßmodellierung

Im folgenden werden die Modellierungskonzepte erläutert, die *genform* zur Beschreibung von Prozessen anbietet: In der Prozeß-Modellierungssprache (PML) von *genform* finden sich zahlreiche Konzepte und Ideen anderer Ansätze wieder (mehr zu Unterschieden und Gemeinsamkeiten in Abschnitt 4.2.7). Bei der Entwicklung der *genform*-PML standen folgende Entwurfsziele im Vordergrund:

1. Die Sprache soll ohne großen Einarbeitungsaufwand verwendbar sein und Prozesse für Planer, Manager und Entwickler übersichtlich darstellen und veranschaulichen. Prozesse werden daher graphisch dargestellt. Für eine automatisierte Prozeßunterstützung wird die graphische Darstellung um Implementierungsdetails ergänzt.

2. Mit *genform* soll eine Werkzeugunterstützung für einen gegebenen Prozeß gebaut und die Werkzeuge mit dem Prozeß integriert werden. Die PML muß also Konzepte zur Beschreibung von Werkzeugen und ihrer Koordination enthalten.
3. Planungs- und Managementwerkzeuge und die Prozeßmaschine werden aus Komponenten des *genform*-Frameworks konstruiert. Die Werkzeuge nutzen also das Repository zur Datenverwaltung. Der Prozeßzustand, der sich aus der Interpretation des Prozeßmodells ergibt, muß also im Repository verwaltet und von den generischen Werkzeugen bearbeitet werden können.
4. Die Datenmodelle, mit denen die Struktur von Dokumenten beschrieben wird, sind feingranular. Alle Projektdaten werden in einer komplexen Struktur verwaltet, die ein azyklischer Graph aus Komponentenbeziehungen aufspannt. Mit der PML müssen also Beziehungen zwischen Aufgabentypen und den Typen aus dem Dokumentdatenmodell beschrieben werden können.

4.2.1 Überblick

Prozeßmodelle werden in *genform* graphisch dargestellt. Abbildung 4.2 zeigt als Beispiel einen Ausschnitt aus dem Wasserfall-Modell.

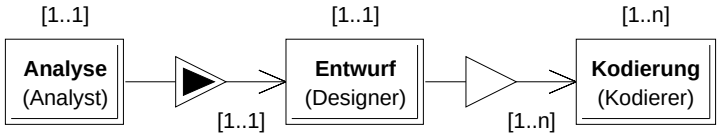


Abbildung 4.2: Ausschnitt aus dem Wasserfall-Modell

Die entwickelte PML stellt keinen wesentlichen Forschungsbeitrag dar. Lediglich die folgenden Eigenschaften sind im betrachteten Kontext interessant und sind Voraussetzung für die Nutzung der Sprache mit *genform*.

Aufgaben und Beziehungen. Das abgebildete Prozeßmodell enthält die Aufgabentypen Analyse, Entwurf und Kodierung. Die Pfeile zwischen den Aufgabentypen symbolisieren Kontrollfluß-Beziehungen. Sie bestimmen, in welcher Reihenfolge die Aufgaben ausgeführt werden können. Durch die Aufgabentypen und ihre Beziehungen wird die Struktur des Prozesses vorgegeben, also erlaubte Folgen von Aufgaben und erlaubte Aufgabenzerlegungen, wie sie alle PML in meist mächtigerer Form anbieten.

Entscheidend für *genform* ist, daß aus dem Prozeßmodell ein Datenbankschema erzeugt und dieses genutzt wird, um die Konsistenz der Prozeßinstanz zu sichern. Die Werkzeuge, mit denen die Prozeßinstanz manipuliert wird, interpretieren das Datenbankschema. Mehr zu Aufgabentypen und ihren Beziehungen in den Abschnitten 4.2.2 und 4.2.3.

Dokumente. Ziel eines Software-Prozesses ist es, Software-Dokumente zu erstellen und zu bearbeiten. Somit sollten die Typen der manipulierten Dokumente und ihre Eigenschaften im Prozeßmodell berücksichtigt werden.

Die Besonderheit in *genform* ist, daß Dokumente feingranular modelliert und mit speziell konfigurierten Werkzeugen bearbeitet werden. Die PML muß also Möglichkeiten bieten, um

- den Aufgaben beliebige Dokumentteile zuordnen zu können und den Bearbeitern einer Aufgabe den Zugriff auf diese Dokumentteile zu erleichtern. Die Dokumentteile können dabei nicht an beliebiger Stelle zwischengespeichert werden, etwa wie Dateien im Arbeitsverzeichnis des Benutzers, sondern bleiben in der Dokumentstruktur.
- die Zugriffsrechte an Dokumenten und Dokumentteilen festzulegen und dabei das komplexe Rechtemodell von PCTE zu berücksichtigen.
- den Aufgaben passend konfigurierte Werkzeuge zuzuordnen, die die Bearbeiter der Aufgabe auf den zugeordneten Dokumenten nutzen. Die Konfiguration eines Werkzeugs wird durch ein Werkzeugschema beschrieben.
- die Zustände von Dokumentteilen zu beschreiben – die möglichen Zustände eines Dokuments und die erlaubten Übergänge können auch von der Aufgabe abhängen, in der das Dokument gerade bearbeitet wird, sollten also nicht im Datenmodell der Dokumente definiert werden.

Mehr zur Verwaltung von Dokumenten in Abschnitt 4.2.4.

Realisierung der Prozeßmodellierungssprache

An dieser Stelle soll kein Beitrag zur Diskussion geleistet werden, ob eine komplexe Sprache zur Beschreibung von Prozeßmodellen einem Ansatz mit mehreren Teilsprachen vorzuziehen ist [63]. In *genform* werden einfache Sprachen oder Beschreibungsmittel verwendet, um die Aspekte eines Prozesses zu beschreiben, die für Experimente mit *genform*-Umgebungen relevant sind. Die Zusammenfassung der Beschreibungsmittel zu *einer* Sprache wird hier nicht untersucht. Die Verwendung mehrerer Teilsprachen hat den Vorteil, daß die Interpretation der Sprachen bei der Prozeßausführung leicht realisiert werden kann. Die Beschreibungsmittel für Prozeßmodelle in *genform* sind:

1. Ein *graphisches Prozeßschema*, das die Aufgabentypen mit ihren Beziehungen, sowie Dokumentordner und Datenflüsse zwischen Ordnern darstellt. Beispiele finden sich in den folgenden Abschnitten. Mit der graphischen Sprache können Prozeßmodelle übersichtlich und leicht verständlich dargestellt werden; auf eine große Mächtigkeit der Sprache wurde zugunsten einer leichten Anwendbarkeit verzichtet. Das Prozeßschema kann leicht in ein Datenbankschema umgesetzt werden, das wiederum zur Verwaltung der Prozeßinstanz dient.
2. *Zustandsdiagramme*, mit denen die Zustände von Aufgaben und Dokumenten beschrieben werden. Alternativ könnte hier auch jede andere Beschreibungsform für Zustände gewählt werden. Zustandsdiagramme haben den Vorteil, daß bereits ein graphischer Editor für sie existiert, daß sie einfach zu handhaben sind, und daß Bedingungen und Aktionen an Zustandsübergängen leicht zugeordnet werden können. Mehr dazu in Abschnitt 4.2.5.
3. *Java-Quelltext*, der die eigentliche Logik des Prozesses, also die Überwachung und Steuerung des Prozeßfortschritts, implementiert. Der Java-Code steckt in den Komponenten der Prozeßmaschine, den Prozeßagenten. Die Anpassung der Prozeßmaschine ist also möglich durch die Auswahl geeigneter Komponenten, durch die Parametrisierung der Komponenten und durch die Erweiterung und Anpassung der Komponenten. Die Komponenten sind

Teil den *genform*-Frameworks. Als Parameter dienen das Prozeßschema und die Zustandsdiagramme.

Der Nachteil bei der Verwendung mehrerer Sprachen ist, daß der Benutzer die verschiedenen Sprachen und ihren Zusammenhang kennen muß. Außerdem muß er wissen, welche Änderung in welcher Sprache nötig ist, um das gewünschte Ergebnis zu erhalten. Andererseits sind die hier verwendeten Sprachen auf ihren jeweiligen Einsatzbereich abgestimmt, weit verbreitet (Zustandsdiagramme, Java) oder einfach zu erlernen (Prozeßschemata). Vor allem stand im Vordergrund:

- die Möglichkeit zur Wiederverwendung vorhandener Werkzeuge für Zustandsdiagramme.
- die Möglichkeit zum kostengünstigen Bau von Werkzeugen für das Prozeßschema.
- die flexible Erweiterbarkeit durch die direkte Verwendung von Java, was gerade beim Bau von Prototypen wichtig ist.

4.2.2 Aufgaben

Aufgaben zerlegen einen Prozeß in einzelne Arbeitsschritte – in Abbildung 4.2 die Schritte Analyse, Entwurf und Kodierung. Jeder Arbeitsschritt sollte ein festgelegtes Ziel haben und eine überschaubare Menge von Tätigkeiten enthalten, so daß die Erledigung der Aufgabe im vorhinein geplant und im nachhinein überprüft werden kann.

Im Prozeßmodell werden die *Typen* der Aufgaben beschrieben. Ein Aufgabentyp legt die Eigenschaften konkreter Aufgaben (der Instanzen des Aufgabentyps) fest. Im alltäglichen Sprachgebrauch wird zwischen Aufgaben*typen* (Java-Implementierung) und konkreten Aufgaben ('Implementiere Klasse X') selten unterschieden. Das gilt auch für viele Prozeßmodellierungsansätze, da Aufgaben als Kopien vordefinierter Aufgaben betrachtet und deren Eigenschaften nachträglich verändert werden.

In *genform* wird das Prozeßmodell als Datenbankschema betrachtet, das die Struktur möglicher Prozesse beschreibt. Eine Aufgabe wird durch Instantiierung eines Aufgabentyps erzeugt. Eine Aufgabe hat folgende Eigenschaften:

- Einen *Namen.*
- Eine *Rolle*, die ein Benutzer besitzen muß, um die Aufgaben des Typs ausführen zu können. Eine Rolle entspricht einer Menge von Kenntnissen und Fähigkeiten (Analytiker, Designer, Kodierer). Die Rolle wird im Symbol des Aufgabentyps angezeigt.
- Eine optionale *Kardinalität.* Sie legt eine untere und obere Grenze für die Anzahl der Instanzen des Aufgabentyps im Prozeß fest: Im klassischen Wasserfall-Modell gibt es genau eine Analyse- und eine Entwurfsaufgabe (Kardinalität [1..1]). Im Modell in Abbildung 4.2 können aber mehrere Implementierungsaufgaben erzeugt werden (Kardinalität [1..n]). Kardinalitäten werden in eckigen Klammern oberhalb des Aufgabensymbols notiert.
- Eine Menge von *Unteraufgaben.* Sie ermöglichen die hierarchische Zerlegung von Aufgaben in Teilaufgaben. Die Teile werden über *Teilaufgaben-Beziehungen* mit dem Ganzen verbunden. Die Beziehungen werden durch Rauten (in Anlehnung an das Aggregationssymbol in UML) dargestellt. *Kardinalitäten* an den Beziehungen definieren untere und

obere Schranken für die Zahl der Teilaufgaben. Beispiel 4.1 verdeutlicht die Aufgabenzerlegung.

- Eine Menge von *Nachfolgern*. Durch *Nachfolgerbeziehungen* (symbolisiert durch Dreiecke, die zum Nachfolger weisen) wird der Kontrollfluß im Prozeß beschrieben. Eine ausführliche Beschreibung findet sich in Abschnitt 4.2.3.
- Eine Menge von *Dokumentordnern*. Sie verwalten die für eine Aufgabe relevanten Dokumente (s. Abschnitt 4.2.4). Ein Dokumentordner enthält Dokumente genau eines Typs. Er dient dem Bearbeiter einer Aufgabe als Ablage für Eingabe- und Ausgabedokumente. Im Wasserfall-Modell werden Analysedokumente, Entwurfsspezifikationen und Quelltexte in den verschiedenen Arbeitsschritten produziert und konsumiert.
- Eine Menge von *Werkzeugschemata*. Jedes Werkzeugschema beschreibt ein Werkzeug, das der Bearbeiter einer Aufgabe nutzen kann. Genauer sind die Werkzeugschemata den Dokumentordnern zugeordnet – sie legen also die Eigenschaften der Werkzeuge zur Bearbeitung der enthaltenen Dokumente fest.

Beispiel 4.1 (Aufgabenzerlegung) Abbildung 4.3 zeigt die Zerlegung der Aufgabe Kodierung in die Unteraufgaben DB-Zugriff, GUI und Integration. Ob der DB-Zugriff implementiert werden muß, hängt von der Art des Systems ab – die Aufgabe ist daher optional (Kardinalität [0..1]). Das GUI kann in mehreren Teilaufgaben implementiert werden [1..n]; danach

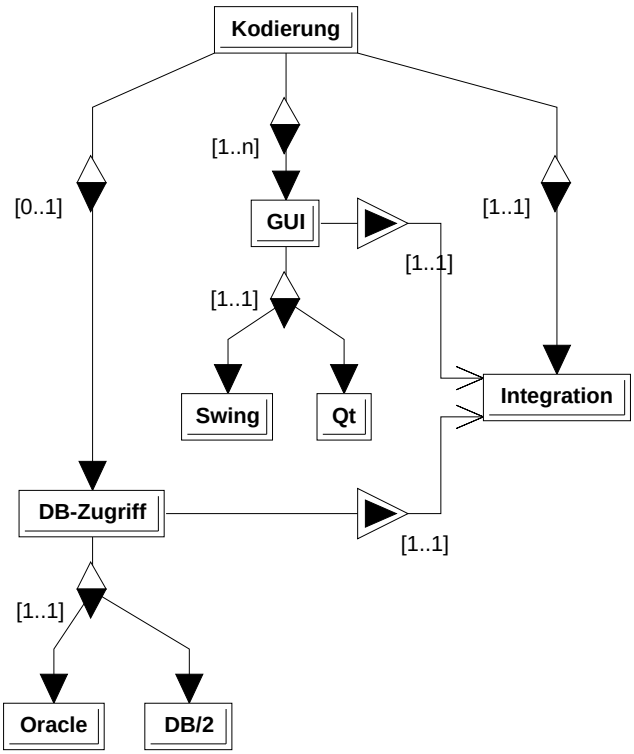


Abbildung 4.3: Beispiel für eine Aufgabenzerlegung

folgt genau eine Integrationsaufgabe. Die Implementierungsaufgaben sind in je zwei Teilaufgaben zerlegt, die die jeweiligen Implementierungsspezifika berücksichtigen (Oracle oder

DB/2 bzw. Swing oder Qt). Wegen der Kardinalität [1..1] muß hier genau eine der Alternativen ausgeführt werden. Auf die Nachfolgerbeziehungen (gefüllte Dreiecke) geht der nächste Abschnitt ein.

□

4.2.3 Kontrollfluß-Beziehungen

Kontrollfluß-Beziehungen beschreiben Abhängigkeiten zwischen Aufgaben und schränken damit die mögliche Reihenfolge bei der Ausführung von Aufgaben ein:

Beispiel 4.2 (Arten von Abhängigkeiten) Im Prozeßmodell aus Abbildung 4.4 soll der Entwurf der verschiedenen Pakete erst beginnen, wenn die Analyse abgeschlossen ist. Sobald der Entwurf beginnt, startet die Dokumentation. Die Implementierung eines Pakets beginnt, sobald es vollständig entworfen wurde; die Entwurfs- und Implementierungsaufgaben werden also parallel ausgeführt.

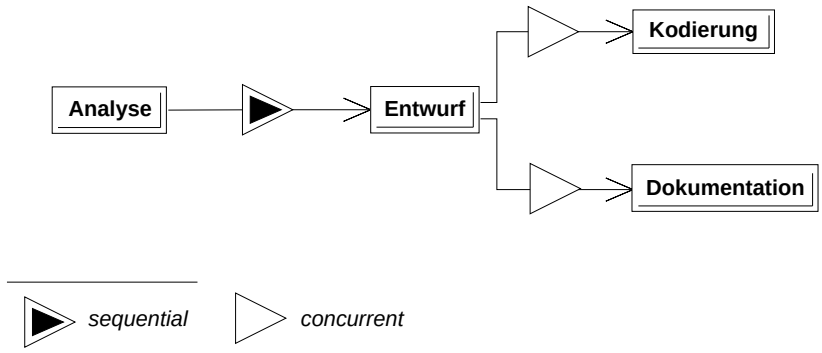


Abbildung 4.4: Beispiel-Prozeßmodell zur Verdeutlichung der Abhängigkeiten zwischen Aufgaben

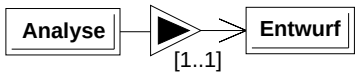
□

In Beispiel 4.2 kommen die drei Arten von Abhängigkeiten zwischen Aufgaben vor, die mit *genform* modelliert werden können:

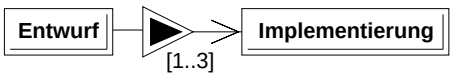
1. Eine *Ende-Anfang-Beziehung* zwischen Analyse und Entwurf (schwarz ausgefüllter Pfeil): Der Nachfolger startet, sobald der Vorgänger abgeschlossen ist – die Aufgaben werden in einer Sequenz ausgeführt.
2. Eine *Anfang-Anfang-Beziehung* zwischen Entwurf und Dokumentation: Der Nachfolger (besser: die abhängige Aufgabe) beginnt, sobald der Vorgänger beginnt (Pfeil nicht ausgefüllt).
3. Eine *feingranulare Abhängigkeit* [191] ermöglicht den Start des Nachfolgers abhängig vom Zustand des Vorgängers – sie stellt also eine Verfeinerung der Anfang-Anfang-Beziehung dar. Ein Beispiel ist die Beziehung zwischen Entwurf und Kodierung: Die Kodierung beginnt, sobald in der Entwurfsaufgabe Entwurfsdokumente produziert wurden. In der graphischen Darstellung werden feingranulare Abhängigkeiten nicht explizit gekennzeichnet.

Bei Ende-Anfang- und Anfang-Anfang-Beziehungen wird abhängig vom Zustand des Vorgängers entschieden, ob der Nachfolger beginnen kann. Bei einer feingranularen Abhängigkeit müssen weitere Bedingungen berücksichtigt werden. Die Bedingungen müssen separat formuliert werden – in *genform* geschieht dies bei der Implementierung der Aufgaben (siehe Abschnitt 5.1.1). Eine Nachfolgerbeziehung hat folgende Eigenschaften:

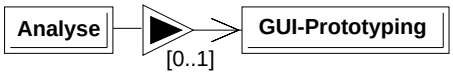
- Die *Beziehungsart*. Sie legt fest, ob es sich um eine Ende-Anfang- oder eine Anfang-Anfang-Beziehung handelt. Im ersten Fall ist die Beziehungsart *sequential* (die beiden Aufgaben werden nacheinander ausgeführt); im zweiten Fall ist sie *concurrent* (die Aufgaben werden gleichzeitig ausgeführt). Existiert eine feingranulare Abhängigkeit, wird in der Implementierung des Nachfolgers festgelegt, wann dieser beginnen kann. In diesem Fall ist die Beziehungsart nicht relevant.
- Eine *Kardinalität*. Sie legt die untere und obere Grenze für die Anzahl der Beziehungen fest, die von einer Aufgabe ausgehen können. Die Kardinalität wird in eckigen Klammern unterhalb des Beziehungssymbols notiert. Abbildung 4.5 zeigt Beispiele für die Verwendung von Kardinalitäten.



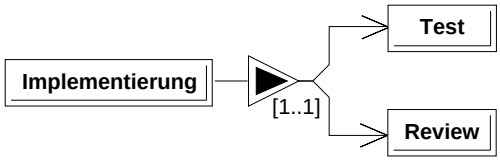
(a) Genau ein Nachfolger



(b) Aufspaltung auf maximal drei Nachfolger



(c) Optionaler Nachfolger

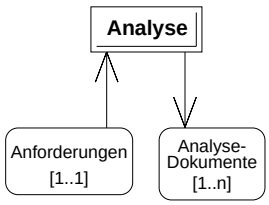


(d) Alternative Nachfolger

Abbildung 4.5: Beispiele für Kardinalitäten

4.2.4 Dokumentordner und Datenflüsse

Der Bearbeiter einer Aufgabe geht von bestimmten Dokumenten aus (etwa einer Anforderungsspezifikation oder einer Entwurfsklasse) und erzeugt neue Dokumente (Analysedokumente) oder ändert bestehende (indem er das Klassengerüst um die Implementierung der Operationen ergänzt). Die bearbeiteten Dokumente kann man als Parameter der Aufgabe betrachten. Die 'Signatur' eines Aufgabentyps wird durch die Zuordnung von (Typen von) *Dokumentordnern* spezifiziert. Abbildung 4.6 verdeutlicht dies: Links die graphische Darstellung einer Aufgabe mit Dokumentordnern; rechts die Deklaration der Aufgabe in Pseudo-Code. Ein Dokumentordner entspricht einem mengenwertigen Parameter. Die Kardinalität des Dokumentordners wird im Ordnersymbol notiert. Sie gibt die untere und obere Grenze



procedure Analyse (
Anforderungen : **IN ARRAY [1..1] OF** Anforderungsdef;
Analyse-Dokumente : **OUT ARRAY [1..n] OF** Klassendiagramm)

Abbildung 4.6: Dokumentordner definieren die Signatur einer Aufgabe

für die Zahl der enthaltenen Dokumente an. Aus Sicht des Benutzers ist ein Dokumentordner eine Arbeitsmappe mit den Dokumenten, die für die Aufgabe relevant sind. Der Benutzer greift auf die Eingabedokumente lesend zu und erzeugt oder verändert Ausgabedokumente.

Technisch ist ein Dokumentordner ein Verzeichnis, das *Verweise* auf Dokumente enthält. Die Dokumente werden in der Dokumentstruktur verwaltet. Die Dokumentstruktur wird im Datenmodell der Dokumente beschrieben. Der Dokumentordner enthält lediglich 'Zeiger', über die direkt auf die Dokumente zugegriffen werden kann. Da Dokumente feingranular modelliert werden, können auch beliebige Dokumentteile (einzelne Klassen eines Diagramms oder einzelne Operationen einer Klasse) in Dokumentordnern verwaltet werden. In Abbildung 4.7 ist ein Dokumentordner mit den Verweisen in die Dokumentstruktur skizziert.

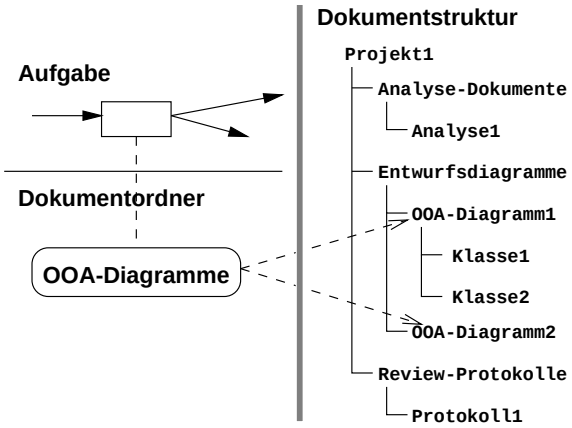


Abbildung 4.7: Dokumentordner mit Verweisen auf OOA-Diagramme

Verzeichnisse. Ein Dokumentordner kann zunächst beliebige Dokumente des vorgegebenen Typs verwalten. Um eine Vorauswahl zu treffen, kann dem Ordner ein Verzeichnisobjekt (kurz: *Verzeichnis*) zugeordnet werden. Durch die ausgehenden Links des Verzeichnisobjekts wird eine Objektmenge definiert, aus der Objekte in den Ordner eingefügt werden können. Mögliche Verzeichnisse wären ein Paket (in den Ordner können nur Klassen des Pakets und seiner Subpakete eingefügt werden) oder ein ER-Diagramm, dessen Einträge bearbeitet werden sollen.

Ein Anwender, der über einen Ordner auf Dokumente zugreift, muß sich also nicht darum kümmern, wo sich die Dokumente in der Projekt- und Dokumentstruktur befinden. Neben

dem Zugriff auf vorhandene Dokumente können Werkzeuge zusätzlich Funktionen anbieten, mit denen

- Einträge direkt im Verzeichnis erzeugt werden können: Ein Dokument wird also an der richtigen Stelle in der Dokumentstruktur (etwa eine Klasse im gewählten Paket) erzeugt und gleichzeitig in den Ordner eingefügt. Neben der Aggregationsbeziehung zwischen dem Ordner und dem neuen Eintrag muß dazu auch eine Kompositionsbeziehung zwischen dem Verzeichnis und dem neuen Eintrag erzeugt werden.
- Verzeichniseinträge über den Ordner gelöscht werden können. Das Löschkommando entfernt also nicht nur das Dokument aus dem Ordner, sondern löscht es auch aus dem Verzeichnis. Somit könnte eine nicht mehr benötigt Klasse auch gleich aus dem bearbeiteten Paket gelöscht werden.

Ob diese Funktionen ausführbar sind, wird durch Zugriffsrechte am Verzeichnisobjekt geregelt.

Beispiel 4.3 (Paket als Verzeichnis) In mehreren Entwurfsaufgaben sollen Analyseklassen verfeinert werden. Die Klassen befinden sich in mehreren Paketen, für jedes Paket wird eine Entwurfsaufgabe angelegt. Jede Entwurfsaufgabe besitzt einen Dokumentordner, in dem der Entwickler die für ihn relevanten Klassen ablegt. Das Verzeichnis des Ordners ist das Paket. Über den Dokumentordner kann der Entwickler direkt auf die von ihm bearbeiteten Klassen zugreifen, muß also nicht durch die Paketstruktur navigieren. Der Entwickler kann auch neue Klassen im Ordner erzeugen – die neuen Klassen sind Komponenten des Pakets und werden in den Dokumentordner eingetragen. Zum Löschen stehen zwei Kommandos zur Verfügung: Zum Entfernen einer Klasse aus dem Ordner und zum gleichzeitigen Löschen der Klasse aus dem Diagramm.

□

Der Typ eines Dokumentordners legt folgende Eigenschaften fest:

- Einen *Namen*.
- Einen *Dokumenttyp*. Nur Dokumente dieses Typs können im Ordner verwaltet werden. Der Dokumenttyp wird über den Typnamen aus dem Dokument-Datenmodell identifiziert.
- Einen *Verzeichnistyp*. Er legt den Typ der Dokumente fest, die als Verzeichnis dienen können.
- Einen Zugriffsmodus für die enthaltenen Dokumente. Der Zugriffsmodus ähnelt der Übergabeart von Parametern einer Prozedur. Er legt fest, ob es sich bei den enthaltenen Dokumenten um Eingabedokumente ('IN') oder Ausgabedokumente ('OUT') handelt, oder ob vorhandene Dokumente in der Aufgabe bearbeitet werden ('INOUT'). Und er bestimmt die Zugriffsrechte, die dem Bearbeiter einer Aufgabe an den enthaltenen Dokumenten gewährt werden: Auf Eingabedokumente kann nur lesend zugegriffen werden; Ausgabedokumente können erzeugt, gelesen und geschrieben werden; zur Bearbeitung vorhandener Dokumente müssen Schreib- und Leserechte gewährt werden. Der Zugriffsmodus wird durch die Pfeilspitzen an der Beziehung zwischen Aufgabentyp und Dokumentordner symbolisiert. Außerdem können für einen Ordner weitere Zugriffsrechte spezifiziert werden. Sie regeln den Zugriff anderer Benutzer auf die Dokumente im Ordner.

– Eine *Kardinalität*, also die Unter- und Obergrenze für die Zahl der enthaltenen Dokumente.
– Eine Menge von *Werkzeugschemata*. Diese beschreiben die Werkzeuge, die der Bearbeiter einer Aufgabe auf den Dokumenten im Ordner nutzen kann. Ein Werkzeugschema legt die Sicht der Bearbeiter einer Aufgabe auf die Dokumente fest und die Darstellungen und Kommandos im Werkzeug. Die Menge von Dokumenten und Werkzeugen, die einer Aufgabe zugeordnet sind, wird auch als *Arbeitskontext* (*work context*) bezeichnet [323].
– Ein *Zustandsdiagramm* beschreibt die Zustände und Zustandsübergänge der enthaltenen Dokumente.
– *Datenflußbeziehungen* zu anderen Dokumentordnern. Über Datenflußbeziehungen können Dokumente zwischen Ordnern (und damit zwischen Aufgaben) weitergeleitet werden.

Datenmodell

In Abbildung 4.8 ist das Datenmodell für Dokumentordner dargestellt. Zu unterscheiden ist der allgemeine Teil mit den Basistypen für Aufgaben, Dokumentordner und Dokumente und der prozeßspezifische Teil mit Subtypen für den konkreten Prozeß – hier gemäß Beispiel 4.3 die Entwurfsaufgabe, der Dokumentordner für Entwurfsklassen und die Klassen selbst. Abbildung 4.9 zeigt die Objekte und Links in der Objektbank.

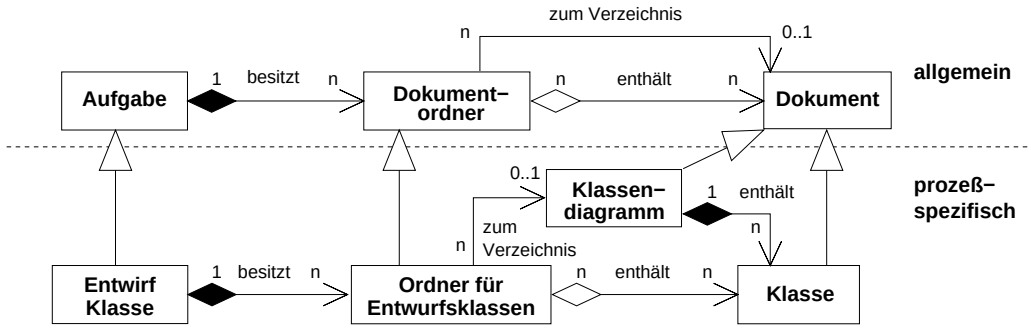


Abbildung 4.8: Datenmodell von Dokumentordnern

Zugriffsrechte

Zugriffsrechte schränken die möglichen Änderungen ein, die der Bearbeiter einer Aufgabe durchführen kann. Folgende Rechte sind Teil der Definition von Ordnertypen:

– Die Rechte an den Einträgen im Ordner werden durch die Übergabeart festgelegt: Leserecht an Eingabedokumenten ('IN'), Schreibrecht an Ausgabedokumenten ('OUT').
– Die Rechte am Ordner legen fest, ob der Bearbeiter Einträge hinzufügen oder löschen und somit die Menge der bearbeiteten Dokumente selbst festlegen darf. Dies ist sinnvoll bei der Überarbeitung eines Systems, wenn im vorhinein noch nicht feststeht, welche Klassen geändert werden müssen. Die Menge der Klassen, die in einer Testaufgabe geprüft werden müssen, sollte hingegen fest sein.
– Über Zugriffsrechte am Verzeichnis kann es dem Bearbeiter erlaubt werden, neue Dokumente zu erzeugen oder Dokumente aus der Dokumentstruktur zu löschen – dies wäre bei größeren Änderungen notwendig, die auch eine Änderung der Klassenstruktur erfordern.

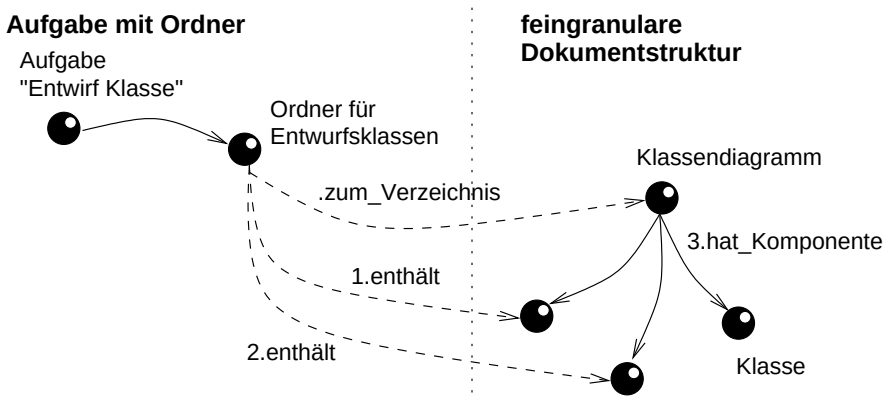


Abbildung 4.9: Dokumentordner in der Objektbank

Dokumentordner vs. Arbeitsbereiche

Es bestehen Ähnlichkeiten zwischen Arbeitsbereichen [107], wie sie Versionsverwaltungssysteme anbieten, und Dokumentordnern: Beide enthalten die für eine Aufgabe relevanten Dokumente und beide ermöglichen den direkten Aufruf passender Werkzeuge. Primäres Ziel der Arbeitsbereiche ist es allerdings, die Versionierung von Dokumenten zu unterstützen – diese fehlt bei den Dokumentordnern.

Bezogen auf die Versionierung haben Arbeitsbereiche zwei Aufgaben [323]: Sie verbergen die Versionen der Dokumente vor den Werkzeugen – die Werkzeuge arbeiten auf einer Dokumentkopie im Arbeitsbereich und müssen sich nicht um die Versionierung kümmern. Zum anderen unterstützt der Arbeitsbereich auch die Kooperation mehrerer Beteiligter und die parallele Bearbeitung von Dokumenten. Jeder Entwickler arbeitet auf Kopien der Dokumente in seinem privaten Arbeitsbereich und muß beim Im- und Export von Dokumenten festgelegte Regeln beachten.

In *genform* wird die Kooperation zwischen Entwicklern über Zugriffsrechte und Sichten geregelt: Sie legen fest, welche Änderungen ein Entwickler an einem Dokument durchführen kann. Dokumente werden allerdings nicht kopiert, sondern ein Ordner enthält Verweise auf die Originale in der Dokumentstruktur. Eine parallele Bearbeitung ist möglich, wenn mehrere Entwickler Schreibrechte auf einem Dokument haben. Durch den Benachrichtigungsmechanismus erhält jeder Entwickler eine aktuelle Sicht auf das Dokument. Die feingranularen Sperren von H-PCTE verhindern Änderungsanomalien, ohne die Parallelität unnötig einzuschränken.

Die meisten Ansätze zur Versionsverwaltung gehen von einer grobgranularen Modellierung der Dokumente aus: Ein Dokument entspricht einer Datei, die als Ganzes kopiert, versioniert und vom Werkzeug bearbeitet wird. Die Versionierung feingranularer Dokumente wird in einer anderen Arbeit [264] untersucht. Dort wird versucht, das entwickelte Konfigurationskonzept mit *genform* zu integrieren, so daß Dokumentordner Versionen von Dokumenten verwalten. Dokumente könnten somit in verschiedenen Aufgaben parallel bearbeitet werden und es stünden Mechanismen zum Erzeugen von Versionen, sowie zum Erkennen und Mischen von Änderungen zur Verfügung.

Datenflüsse

Datenflußbeziehungen dienen zur Weiterleitung von Dokumenten zwischen Aufgaben. Sie haben Ähnlichkeit mit den Datenflußbeziehungen in Datenflußdiagrammen, die einen typisierten Datenaustausch zwischen den Prozessen eines Systems modellieren – Datenflußdiagramme können also als eine Möglichkeit zur Beschreibung von Software-Prozessen betrachtet werden [323].

Datenflußbeziehungen zwischen Dokumentordnern können auch als verfeinerte Kontrollflußbeziehungen interpretiert werden [177]: Eine Aufgabe kann nicht nur abhängig vom 'externen' Zustand eines Vorgängers aktiviert werden (Vorgänger gestartet, beendet, unterbrochen), sondern es kann auch berücksichtigt werden, welche Dokumente der Vorgänger produziert hat, und welchen Zustand diese haben. Joeris et al. [191] sprechen daher auch von "feingranularen Abhängigkeiten" zwischen Aufgaben.

In *genform* besitzt eine Datenflußbeziehung folgende Eigenschaften:

- Eine *Bedingung*, die Dokumente erfüllen müssen, damit sie transportiert werden können. Die Bedingung kann sich auf den Zustand des Dokuments beziehen (definiert im Zustandsdiagramm des Ordners) oder auf eine andere meßbare Eigenschaften (Größe, Anzahl Fehler); letztere muß mit einer speziellen Prüfung ermittelt werden.
- Der *Ausführungsmodus* legt fest, ob Dokumente automatisch weitergeleitet werden, sobald sie die Bedingung erfüllen, oder ob der Benutzer die Dokumente mit Hilfe seines Management-Werkzeugs weiterleiten kann. Der Manager einer Aufgabe kann im zweiten Fall bestimmen, welche Dokumenten wann an welche Aufgaben (und damit welche Bearbeiter) weitergeleitet werden.
- Der *Datenflußtyp* gibt an, was mit den übertragenen Dokumenten im Zielordner passiert: Normalerweise sind Dokumente nach der Übertragung im Zielordner enthalten (Typ *contents*). Hat der Datenfluß den Typ *decompose*, wird das übertragene Dokument zum *Verzeichnis* des Zielordners, die Komponenten des Dokuments also zu dessen Inhalt – beim Übertragen wird das Dokument in seine Bestandteile zerlegt. Ein *recompose* faßt umgekehrt die Komponenten eines Dokuments zusammen: Das Verzeichnis des Ausgangsordners wird zum Dokument im Zielordner. Eine Datenflußbeziehung vom Typ *contents* wird mit einer gefüllten Pfeilspitze dargestellt, während die Pfeilspitzen der beiden anderen Datenflußtypen weiß sind und ein 'D' oder 'R' für *decompose* bzw. *recompose* den Datenflußtyp kennzeichnet.

Das folgende Beispiel verdeutlicht den Zusammenhang zwischen Dokumentordnern, Datenflüssen und dem Datenmodell der Dokumente:

Beispiel 4.4 (Zerlegung eines Dokuments) Abbildung 4.10 zeigt einen Ausschnitt aus einem Prozeßmodell mit den Aufgaben Analyse und Verfeinere Klassen. In der Analyse werden Analysediagramme erzeugt. Die enthaltenen Klassen werden dann in einer oder mehreren Aufgaben um Entwurfsinformationen erweitert. Ein Klassendiagramm wird über die eingezeichnete Datenflußbeziehung an die Verfeinerungsaufgabe weitergeleitet, sofern es sich im Zustand 'geprüft' befindet. Das Diagramm wird dort zum Verzeichnis des Klassen-Ordners. Der Bearbeiter der Verfeinerungsaufgabe kann also direkt auf die Klassen des Diagramms zugreifen. Falls er die nötigen Zugriffsrechte besitzt, kann er auch Klassen in den Ordner

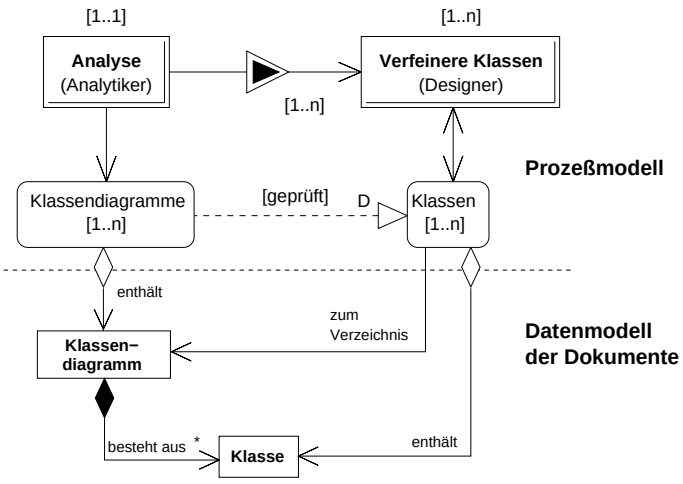


Abbildung 4.10: Dokumentordner mit Datenflußbeziehung vom Typ *decompose*

aufnehmen oder aus dem Ordner entfernen, sowie neue Klassen im Diagramm erzeugen oder aus dem Diagramm löschen.

□

4.2.5 Zustandsdiagramme

Der Zustand des Entwicklungsprozesses spiegelt die aktuelle Projektsituation wider. Der Zustand des Prozesses ergibt sich aus den Zuständen der verschiedenen Teilaufgaben: Manche werden gerade bearbeitet, andere sind abgeschlossen, manche zeitweise unterbrochen. Auch die Zustände der Dokumente sind relevant, um die Projektsituation beurteilen zu können: Dokumente können vorläufig, fertig bearbeitet, getestet, fehlerhaft oder fehlerfrei sein.

Dem Übergang zwischen zwei Zuständen sind zugeordnet: Eine Bedingung, die festlegt, wann der Zustandswechsel durchgeführt werden kann, und eine Aktion, die beim Zustandswechsel ausgeführt wird – Beispiele für solche Aktionen sind die Benachrichtigung des Managers über den Abschluß einer Aufgabe, das Einfrieren einer Dokumentversion nach erfolgreichem Test, oder das Erzeugen eines abgeleiteten Dokuments. Die möglichen Zustände von Aufgaben und Dokumenten werden mit Zustandsdiagrammen beschrieben. Abschnitt 5.2.4 zeigt, wie ein Zustandsautomat für Dokumente realisiert wird.

Zustände von Aufgaben

Für die Verwaltung von Aufgabenzuständen wird das Zustandsdiagramm aus Abbildung 4.11 vorgegeben. Der Prozeßmodellierer kann das vorgegebene Zustandsdiagramm anpassen, indem er Bedingungen und Aktionen definiert oder die Zustände und Übergänge im Diagramm selbst ändert. Beispiele für angepaßte Zustandsdiagramme zeigt Abbildung 4.12: An den Zustandsübergängen sind Bedingungen/Aktionen in gestrichelten Rechtecken notiert. Mit den Zustandsdiagrammen können Zwischenschritte innerhalb einer Aufgabe modelliert werden (Bearbeitung von Paketen und Klassen während der Implementierung in Abbildung

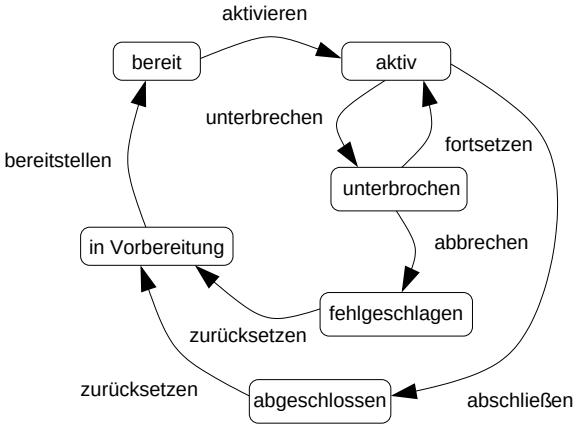
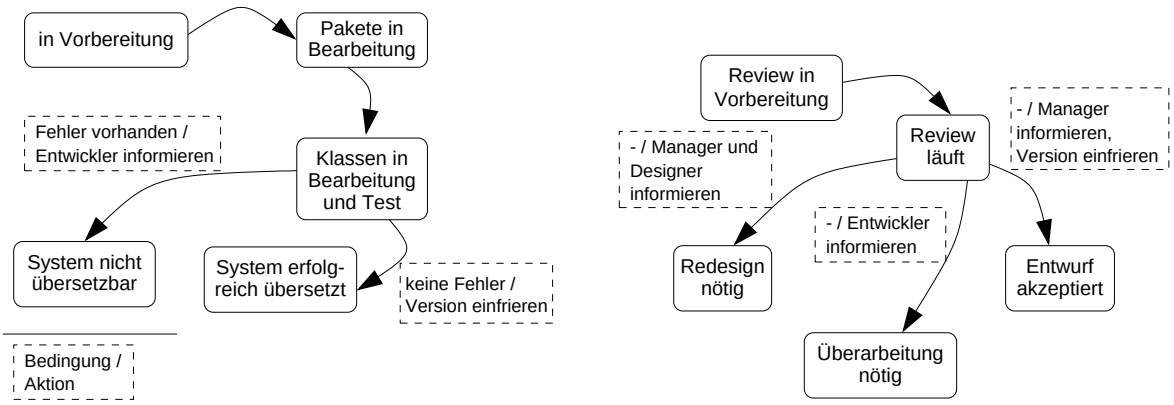


Abbildung 4.11: Vorgegebenes Zustandsdiagramm für Aufgaben



(a) Implementierung (b) Review

Abbildung 4.12: Angepaßte Zustandsdiagramme für Aufgaben

4.12(a)), oder unterschiedliche Ausgänge von Aufgaben verdeutlicht werden (beim Review in Abbildung 4.12(b)).

Zustände von Dokumenten

Auch die Zustände von Dokumenten werden durch Zustandsdiagramme beschrieben. Das vorgegebene Zustandsdiagramm für Dokumente zeigt Abbildung 4.13. Mit den Zustandsdiagrammen werden die Eigenschaften von Dokumenten explizit gemacht und somit die Integration von Management-Prozessen und technischen Prozessen ermöglicht [189]. Die Zustandsdiagramme sind den Dokumentordnern zugeordnet – ein Dokument kann also in verschiedenen Ordnern unterschiedliche Zustände haben: Die Zustandsdiagramme sind *aufgabenspezifisch*. Zwei Beispiele für angepaßte Zustandsdiagramme zeigt Abbildung 4.14.

Dokumentzustände werden auch zur Steuerung der Datenflüsse zwischen Aufgaben genutzt: Ein Dokument kann von einer Aufgabe A an eine Nachfolge-Aufgabe B (genauer: von einem Dokumentordner D_A zu einem Dokumentordner D_B) geleitet werden, wenn es die an der Datenflußbeziehung angegebene Bedingung erfüllt. Diese Bedingung kann sich auch auf den

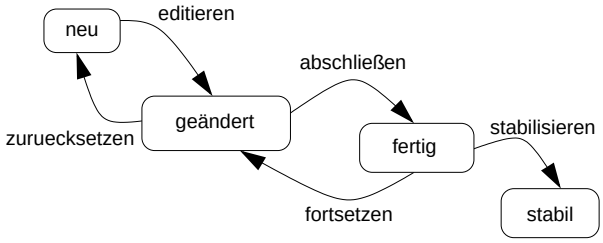


Abbildung 4.13: Vorgegebenes Zustandsdiagramm für Dokumente

(a) Klassendiagramm in der Analysephase

(b) Klasse in der Implementierung

Abbildung 4.14: Aufgabenspezifische Zustandsdiagramme für verschiedene Dokumenttypen

Zustand des Dokuments beziehen, der im Zustandsdiagramm von D_A definiert ist. In D_B besitzt das Dokument zunächst den initialen Zustand. In Bedingungen können beliebige Eigenschaften der Dokumente geprüft werden, also auch solche, die nicht explizit auf Zustände abgebildet sind. Mehr dazu in Abschnitt 5.1.

Ein Beispiel zeigt Abbildung 4.15: Klassen werden von der Implementierungs- an die Test-

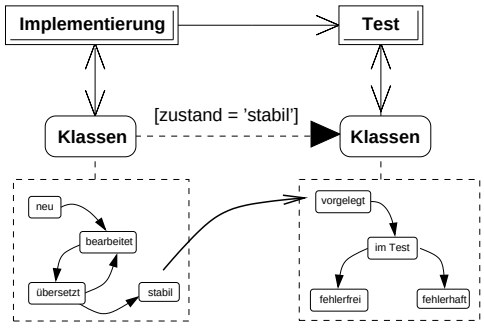


Abbildung 4.15: Aufgaben mit Dokumentordnern und lokalen Zustandsdiagrammen

aufgabe weitergeleitet, wenn sie sich im Zustand 'stabil' befinden; im Ordner der Testaufgabe erhalten sie den Zustand 'vorgelegt'.

Arten von Zustandsübergängen

Um festzulegen, wie die Prozeßmaschine Zustandsübergänge behandelt, werden folgende Arten von Zustandsübergängen unterschieden:

- *Manuelle* Zustandsübergänge (durch M gekennzeichnet) werden vom Benutzer ausgelöst, indem er mit einem Werkzeug einen neuen Zustand auswählt: Alle erlaubten Zielzustände (also Zielzustände von solchen Übergängen, deren Bedingung erfüllt ist) werden angezeigt und der Benutzer wählt einen neuen Zustand aus.
- *Werkzeug-Zustandsübergänge* (*Tool*, T) werden von Werkzeugen (etwa Prüfwerkzeugen) ausgelöst. Ihre Zielzustände werden dem Benutzer also nicht zur Auswahl angeboten (vgl. Abschnitt 5.2.3). Soll ein Werkzeug einen Zustandswechsel durchführen, so muß das Zustandsdiagramm im Werkzeug konsistent sein mit dem Zustandsdiagramm im Prozeßmodell. Auch muß das Werkzeug die Bedingungen der Zustandsübergänge selbst prüfen.
- *Agenten-Zustandsübergänge* (A) werden von der Prozeßmaschine ausgelöst, sobald die Bedingung erfüllt ist. Sie erscheinen also nicht in der Benutzungsschnittstelle (wie die manuellen Zustandsübergänge) und hängen nicht vom Aufruf eines Werkzeugs ab (wie die Werkzeug-Übergänge). Da die Prozeßmaschine die Bedingungen an Zustandsübergängen überwacht, wird ein Agenten-Übergang ausgeführt, sobald die Bedingung wahr ist. Mehr zur Prozeßmaschine in Abschnitt 5.1.

Zu bemerken ist noch, daß Zustandsübergänge durch ihre Zielzustände identifiziert werden, also selbst keine Bezeichner haben. Zustandsdiagramme müssen daher so entworfen werden, daß zu keinem Zeitpunkt mehr als ein Übergang zwischen zwei Zuständen gültig sein kann. Beim manuellen Wechsel können sonst die beiden Übergänge nicht unterschieden werden; bei Agenten-Zustandsübergängen würde zufällig einer der gültigen Übergänge ausgewählt.

4.2.6 Prozeß-Architektur vs. Prozeß-Implementierung

Die graphische Darstellung des Prozesses aus Abbildung 4.2 stellt die *Entwicklungsarchitektur* [117] des Prozesses dar: Die Bausteine des Prozesses und ihre Beziehungen werden auf hoher Abstraktionsebene beschrieben. Auf die Darstellung von 'Implementierungsdetails' des Prozesses wird hier verzichtet.

Die Prozeßmaschine interpretiert das Prozeßmodell. Die Ausführung beruht auf der im Prozeßmodell definierten Architektur und den Implementierungsdetails des Prozesses (Abbildung 4.16). Diese ergeben sich aus der Art, wie die Prozeßmaschine die Architektur interpretiert, somit aus der Implementierung der Prozeßmaschine und aus expliziten Anweisungen zur Steuerung der Prozeßmaschine. Um letztere wird die Architekturbeschreibung ergänzt.

Das Ergebnis eines Interpretationsschrittes ist zum einen ein neuer Prozeßzustand – da der Prozeßzustand im Repository verwaltet wird, bedeutet dies zunächst, daß die persistente Repräsentation des Prozeßzustands verändert wird, was sich wiederum auf den 'realen' Software-Entwicklungsprozeß auswirken kann – etwa, wenn der Entwickler eine neue Aufgabe aus seiner Agenda ausführt oder ein neues Dokument erzeugt und bearbeitet wird. Zum anderen kann ein Interpretationsschritt auch direkte externe Auswirkungen haben, etwa indem Nachrichten an Benutzer gesendet oder Dokumente ausgedruckt werden.

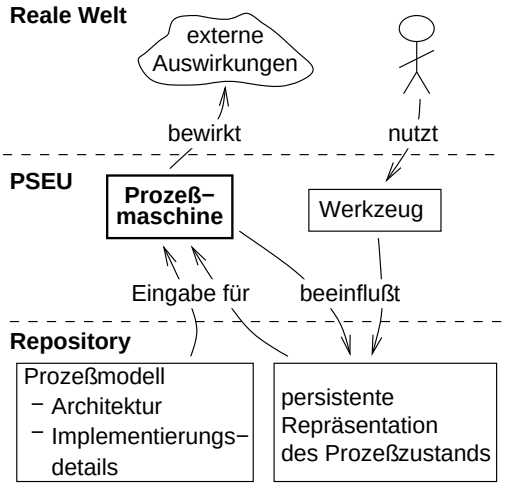


Abbildung 4.16: Interpretation des Prozeßmodells

Um eine Prozeßunterstützung für ein konkretes Projekt zu konstruieren, müssen also die Architektur und die Implementierungsdetails des Prozesses festgelegt werden. Dabei ergänzen sich in *genform* die Spezifikation des Prozesses im Prozeßmodell und die Implementierung der Prozeßmaschine, die ebenfalls prozeßspezifisch angepaßt werden kann – wie dies möglich ist, zeigt Abschnitt 5.1.

4.2.7 Andere Ansätze

Das Meta-Modell von *DYNAMITE* [158] ähnelt sehr stark dem von *genform*: Prozesse werden als Netz von Aufgaben mit Kontroll- und Datenflußbeziehungen beschrieben. Letztere besitzen aber keine Bedingung zur Steuerung der Datenflüsse. Prozesse werden mit Graphschemata beschrieben, in denen die Struktur der Prozesse und mögliche Änderungsoperationen spezifiziert sind. Eine Änderungsoperation entspricht einer Änderung des Prozeßzustands wie dem Hinzufügen einer neuen Aufgabe oder dem Erzeugen einer Nachfolgerbeziehung. Auch können die erlaubten Änderungen vom aktuellen Zustand der Aufgaben abhängen: Eine aktive Aufgabe kann nicht gelöscht werden, einer abgeschlossenen können keine Dokumente zugeordnet werden. Die erlaubte Zahl von Aufgaben wird durch Kardinalitäten eingeschränkt. In *genform* werden zusätzlich Kardinalitäten an Nachfolgerbeziehungen und Teilaufgabenbeziehungen angegeben. Anders als in *genform* ist das Zustandsdiagramm für Aufgaben in *DYNAMITE* fest vorgegeben – nur die Bedingungen an Zustandsübergängen können geändert werden. Dokumente und ihre Zustände oder Werkzeuge werden in *DYNAMITE* nicht berücksichtigt. Prozesse werden wie in *genform* auf zwei Ebenen spezifiziert: Die Struktur des Prozesses wird graphisch dargestellt, das Verhalten von Aufgaben und Beziehungen (*"states, state transitions, conditions, events, and triggers"*) werden separat spezifiziert. *Feedback flows* in *DYNAMITE*, spezielle Kontrollfluß-Beziehungen, die Rückmeldungen an Vorgängeraufgaben übertragen, gibt es in *genform* nicht – hier können normale Kontrollflußbeziehungen mit dem gleichen Zweck verwendet werden.

OIKOS [251] verwaltet nur zwei Dokumentzustände: Ein Dokument kann sich gerade in Bearbeitung befinden oder fertig (aber nicht unbedingt endgültig) sein. Dokumente im ersten

Zustand befinden sich in den Arbeitsbereichen der Entwickler, während fertige Dokumente im Repository liegen.

In den Ansätzen *FUNSOFT* [141] und *SLANG* [21] werden Prozesse mit Petri-Netz-Varianten modelliert. Die typisierten Marken im Netz repräsentieren Produkte, Ressourcen oder Steuerinformationen. Die Datenflußbeziehungen in *genform* ähneln den Transitionen in PrT-Netzen, die Marken auf ihre Ausgabeplätze übertragen, sofern die Marken eine bestimmte Bedingung erfüllen. In *genform* transportieren die Datenflußbeziehungen allerdings nur Dokumente.

In *VPL* [293] gibt es nur eine Art von Beziehung zwischen Aufgaben, die sowohl den Kontroll- als auch den Datenfluß beschreibt. Die Daten sind allerdings nicht typisiert. Welche Daten ausgetauscht und verarbeitet werden, geht daher nur aus den Aufgabennamen hervor (*"design system"*, *"implement module"*).

In den *DesignNets* von Liu und Horowitz [227] sind nur Datenflußbeziehungen zwischen Aufgaben erlaubt, über die Produkte weitergeleitet werden. Die Produkte können dabei in ihre Bestandteile zerlegt und wieder zusammengesetzt werden – dies wird in *genform* durch den Datenflußtyp gesteuert. Neben Aufgaben und Produkten werden auch Ressourcen modelliert. Weder *VPL* noch *DesignNets* berücksichtigen die Werkzeuge, die in einer Aufgabe eingesetzt werden.

In *Serendipity* [148] können Teile einer Prozeßdefinition kopiert und im neuen Prozeß wiederverwendet werden. Das Prozeßmodell enthält Aufgaben mit hierarchischen Verfeinerungen und Beziehungen zwischen Aufgaben. Über letztere werden Ereignisse zwischen den Aufgaben ausgetauscht. Ereignisse werden beispielsweise beim Aktivieren und Deaktivieren einer Aufgabe ausgelöst und können eine abhängige Aufgabe aktivieren. Anders als in *genform* können für die Aufgabenzustände keine Zustandsautomaten definiert werden. Die verwendete Sprache basiert auf der von Swenson entwickelten *Visual Process Language* [300] und erweitert diese um die Möglichkeit, den Aufgaben Dokumente und Werkzeuge zuzuordnen. Die Datenmodelle der Dokumente werden allerdings nicht berücksichtigt, folglich kann nicht auf die Eigenschaften der Dokumente reagiert werden. Auch können keine Dokumentzustände definiert und im Prozeß berücksichtigt werden.

Joeris [190] unterscheidet zwischen dem *kontextspezifischen* und dem *kontextfreien* Verhalten von Aufgabentypen: Das kontextfreie Verhalten wird über Zustandsdiagramme definiert, deren Transitionen mit ECA-Regeln versehen werden; das kontextspezifische Verhalten ergibt sich aus den Kontrollflußbeziehungen zwischen Aufgaben. Dokumente werden über Datenflüsse zwischen Aufgaben weitergeleitet, allerdings gibt es keine Ordner zur Verwaltung von Dokumenten und zur Festlegung der Zugriffsrechte. Auch können den Dokumenten keine Werkzeuge zur Bearbeitung zugeordnet werden. Die Datenmodelle sind grobgranular – die Eigenschaften, innere Struktur und die Dekomposition von Dokumenten kann also nicht direkt berücksichtigt werden.

4.3 Instantiierung des Prozeßmodells

Das Prozeßmodell beschreibt, wie ein konkreter Prozeß ablaufen soll: Welche Arten von Aufgaben von welchen Rollen in welcher Reihenfolge wie ausgeführt werden. In diesem Abschnitt

geht es darum, wie das Prozeßmodell instantiiert und zur Steuerung der Prozeßausführung genutzt wird. Im Englischen wird zwischen *process execution* und *process enactment* unterschieden. Ersteres bedeutet die automatisierte Ausführung des Prozesses, während *enactment* die beteiligten Personen einbezieht, die im Prozeß agieren, Entscheidungen treffen und den Ablauf beeinflussen. In diesem Sinne wird hier mit 'Ausführung' des Prozesses *process enactment* gemeint.

Ein Prozeßmodell wird in drei Schritten instantiiert – dargestellt in Abbildung 4.17 als *genform*-Prozeßmodell.

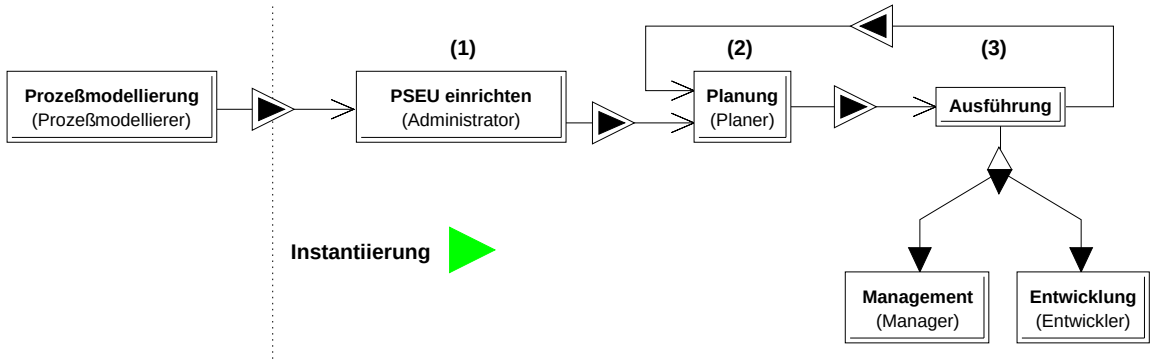
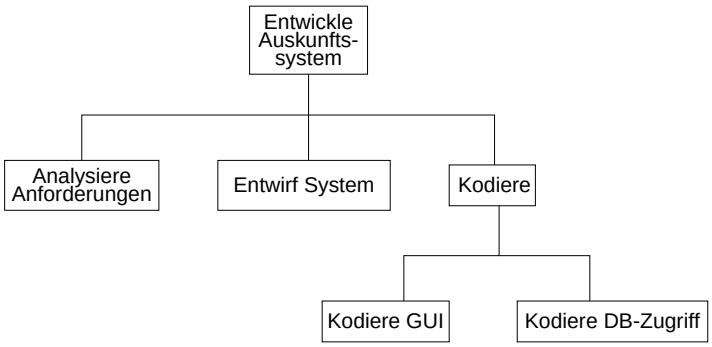


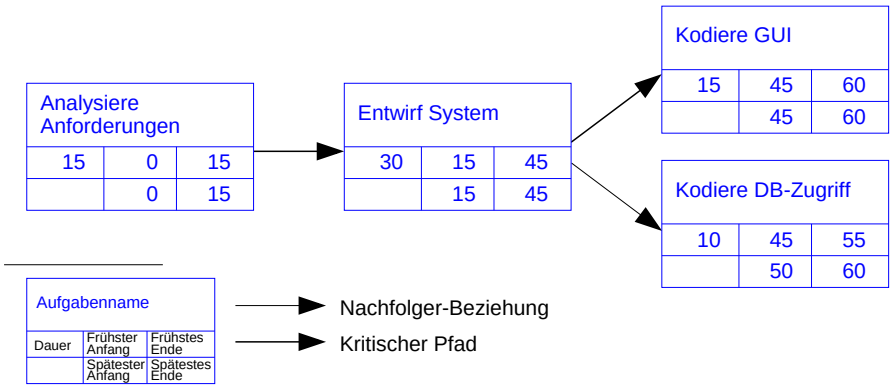
Abbildung 4.17: Ablauf bei der Instantiierung des Prozeßmodells

1. Zunächst wird die PSEU für den Prozeß eingerichtet:
   - Das Repository wird initialisiert, in dem alle Projektdaten zentral verwaltet werden. Es müssen Datenbankschemata, Benutzer und Benutzergruppen eingerichtet und weitere von der PSEU benötigte Daten (Verzeichnisse, organisatorische Daten) erzeugt und mit passenden Zugriffsrechten versehen werden.
   - Es werden ausführbare Werkzeuge zur Verfügung gestellt und die Betriebssystem-Umgebung eingerichtet.
2. Als zweiter Schritt wird der Prozeß *geplant*: Abhängig vom zu entwickelnden Produkt und Randbedingungen wie Terminen und verfügbaren Entwicklern wird der Projektverlauf im vorhinein festgelegt [304]. Die Produkte der Planungsphase sind unterschiedliche Darstellungen des geplanten Projektverlaufs: Abbildung 4.18 zeigt eine *Aufgabenzerlegung* (*work breakdown structure* [227]), einen *Vorgangsknoten-Netzplan* (*Metra Potential Method*, MPM) und ein *Gantt-Diagramm* [17]. Basis für die Planung ist das gewählte Prozeßmodell, das die Arten von Aufgaben und ihre Abhängigkeiten vorgibt.
3. Schließlich wird der Prozeß ausgeführt, also ein Software-Produkt hergestellt oder gewartet. Dabei werden technische Tätigkeiten begleitet von Management-Tätigkeiten, die den Prozeßablauf steuern und überwachen [177].

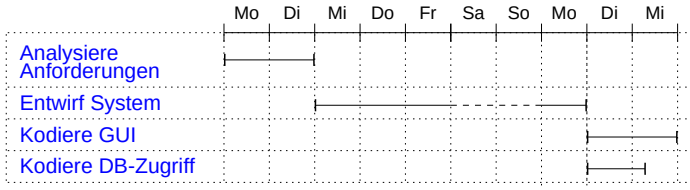
Die Rückkopplungen von der Ausführung zur Planung entstehen, wenn die Planung überarbeitet werden muß – dann wechseln sich Planungs- und Ausführungsphasen ab [132]. Mögliche Risiken, die eine Anpassung der Planung erfordern können, sind [285]: Geänderte Anforderungen, technische Probleme und Fehler; Personalprobleme (falsche, zu viele oder zu wenige Mitarbeiter), sowie ein negatives Projektumfeld (falsche Methoden und Werkzeuge, mangelnde technische Fähigkeiten, mangelnde Projektkontrolle).



(a) Aufgabenzerlegung



(b) Netzplan mit geschätzter Dauer der Aufgaben (in Stunden) und Anfangs- und Endzeitpunkten



(c) Gantt-Diagramm

Abbildung 4.18: Verschiedene Dokumente als Ergebnisse der Planungsphase

Prozeßzustand

Der Zustand des laufenden Prozesses wird im Repository verwaltet. Er enthält Aufgaben und ihre Beziehungen zu anderen Aufgaben, zu Benutzern und zu Dokumenten. Die Aufgaben können verschiedene Zustände haben, wie geplant, in Bearbeitung und abgeschlossen. Auch können weitere Informationen wie Soll-, Istaufwand und Termine an einer Aufgabe gespeichert werden.

In Abbildung 4.19 ist eine Instanz des Wasserfall-Prozeßmodells aus Abbildung 4.2 auf Seite 141 skizziert. Die Grundstruktur bildet ein Graph, dessen Knoten die Aufgaben und dessen Kanten die Beziehungen zwischen Aufgaben sind. Heimann et al. [157] sprechen von einem

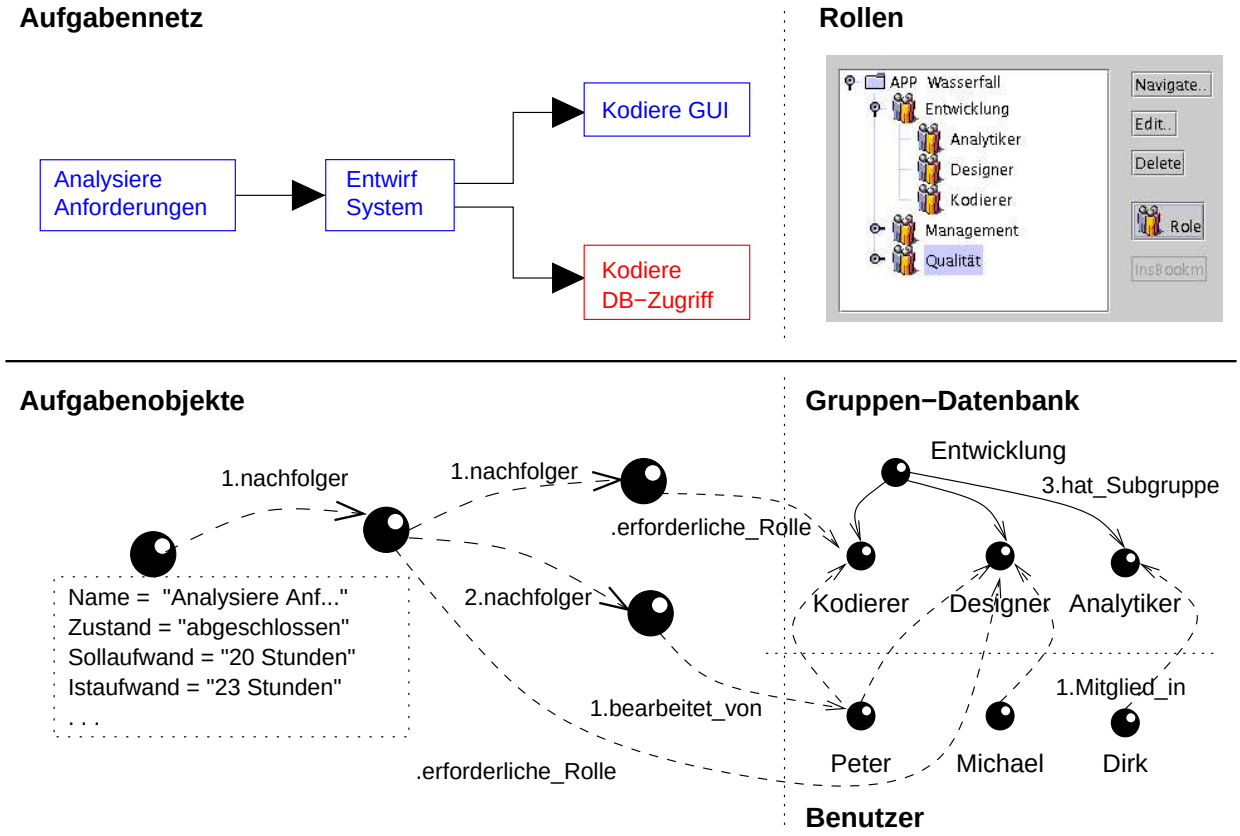


Abbildung 4.19: Instanz des Wasserfall-Modells: Benutzersicht und Repräsentation im OMS

Aufgabennetz. Im oberen Teil von Abbildung 4.19 ist links das Aufgabennetz dargestellt und rechts ein Baum mit Rollennamen. Darunter ist die Repräsentation des Aufgabennetzes im Repository skizziert: Jede Aufgabe wird durch ein Objekt repräsentiert, Nachfolger-Beziehungen durch Links. Im rechten Teil ist die Gruppen-Datenbank dargestellt. Sie enthält ein Objekt für jede Gruppe – hier die Gruppen 'Analytiker', 'Designer' und 'Kodierer' als Untergruppen der Gruppe 'Entwicklung'.

Die Rollen des Prozeßmodells werden auf Benutzergruppen im OMS abgebildet. Über einen Link des Typs erforderliche_Rolle wird einer Aufgabe die Rolle zugeordnet, die der Bearbeiter

besitzen muß. Auch Benutzer werden durch Objekte repräsentiert. Links vom Benutzerobjekt verweisen auf die Gruppen, in denen ein Benutzer Mitglied ist.

Informationen über die Aufgaben werden in Attributen der Aufgabenobjekte verwaltet. Die Informationen werden in verschiedenen Sichten gruppiert – jede Sicht stellt einen bestimmten Aspekt in den Vordergrund. Mehr zu den verschiedenen Sichten auf das Aufgabennetz in Abschnitt 5.2. In den folgenden Abschnitten wird beschrieben, wie eine PSEU mit *genform* eingerichtet wird und wie Planung, Entwicklung und Management unterstützt werden.

4.3.1 PSEU einrichten

Bevor das Repository von der PSEU genutzt werden kann, muß es initialisiert werden: Bei der Initialisierung werden grundlegende Datenstrukturen eingerichtet, die zur Verwaltung der PSEU-Daten nötig sind, und die Mechanismen des OMS so parametrisiert, daß Vorschriften aus dem Prozeßmodell vom OMS automatisch überwacht werden können. Mechanismen zur Integritätssicherung überwachen die korrekte Struktur des Aufgabennetzes, Zugriffsrechte verhindern die unberechtigte Manipulation von Daten. Die Einrichtung des Repositories umfaßt

- das Einrichten von Datenbankschemata – sie beschreiben die Struktur der Daten.
- das Einrichten von Benutzergruppen. Eine Benutzergruppe repräsentiert eine Rolle im Prozeßmodell. Diese Rollen und die Benutzer der PSEU werden direkt vom OMS verwaltet und der Zugriff auf die Objektbank geregelt, indem an die Mitglieder einer Benutzergruppe Zugriffsrechte vergeben werden.
- das Erzeugen von Datenstrukturen, die die PSEU zur Verwaltung der Prozeßdaten benötigt. Dazu gehören die Zustandsdiagramme, mit denen die Prozeßmaschine die Zustände von Aufgaben und Dokumenten verwaltet.

Werkzeugschemata

In *genform* wird eine PSEU durch eine Menge von Werkzeugschemata beschrieben. In Kapitel 3 wurde erläutert, wie mit Werkzeugschemata CASE-Werkzeuge konstruiert werden. Werkzeugschemata werden auch genutzt, um Planungs- und Managementwerkzeuge und die Prozeßmaschine einer PSEU zu spezifizieren. Die Werkzeugschemata werden aber nicht direkt vom Werkzeugentwickler entworfen, sondern aus dem Prozeßmodell abgeleitet. Das Prozeßmodell dient als Vorlage für das Datenmodell der Planungs- und Management-Werkzeuge: Das Datenmodell enthält Objekt- und Linktypen für Aufgaben und ihre Beziehungen. Mit Werkzeugparametern werden die Eigenschaften der Werkzeuge und der Prozeßmaschine beeinflußt.

Für die verschiedenen Sichten auf den Prozeßzustand werden verschiedene Werkzeugschemata erzeugt: Eine Sicht für den Planer mit geschätzten Aufwänden und Kosten, eine Sicht für den Manager mit Aufgabenzuständen und zugeordneten Benutzern und Dokumenten, sowie eine Sicht für den Entwickler, der die ihm zugeordneten Aufgaben mit den relevanten Eigenschaften betrachten kann. Mehr zu den resultierenden Werkzeugen in Abschnitt 5.2.

Benutzergruppen

Die Benutzergruppen des OMS werden zur Repräsentation von Rollen genutzt. Da H-PCTE eine hierarchische Gruppenstruktur ermöglicht (vgl. Abschnitt 2.5.3) können auch Rollen hierarchisch definiert werden. Untergruppen werden als Spezialisierung der Obergruppe betrachtet – etwa 'Java-Kodierer' als Spezialisierung von 'Kodierer'. Die Nutzung der Gruppenstruktur hat folgende Vorteile:

- Die Benutzer und Benutzergruppen werden vom OMS verwaltet. Zur Manipulation der Gruppendatenbank stehen Operationen zur Verfügung, die auch die Konsistenz der Gruppendatenbank sicherstellen.
- Über die aktuelle Gruppe eines Benutzers (*adopted group*) wird auch sein Rechtekontext festgelegt. Entsprechend sorgt das OMS dafür, daß ein Benutzer, der eine bestimmte Rolle übernommen hat, nur erlaubte Operationen ausführen kann.
- Die Mitgliedschaft des Benutzers in verschiedenen Gruppen bestimmt, welche Gruppen für ihn als aktuelle Gruppe in Frage kommen – ein Benutzer kann im Prozeß also nur erlaubte Rollen übernehmen.
- Zum Anlegen der Gruppendatenbank und zum Eintragen von Benutzern können vorhandene Administrationswerkzeuge genutzt werden, die Werkzeuge müssen also nicht neu implementiert werden.

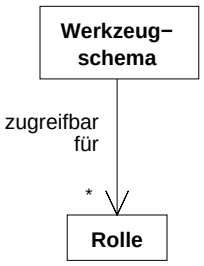
Zum Einrichten der Benutzer und Benutzergruppen sind Administrator-Rechte nötig. Eine Schwäche der Rechteverwaltung von PCTE ist, daß Administrator-Rechte nur für die gesamte Objektbank vergeben werden können – eine Person, die eine PSEU für ein Projekt oder ein Prozeßmodell einrichtet, muß also gleichzeitig *super user* der gesamten Objektbank sein. Sollte dies Probleme bereiten, müssen verschiedene Objektbanken eingerichtet werden mit jeweils lokalen Administratoren.

Mit Zugriffsrechten werden die Modi festgelegt, in denen ein Benutzer auf eine Ressource in der Objektbank zugreifen kann – dazu werden ACL an allen Objekten in der Objektbank gesetzt. Über den Sichtenmechanismus kann der Zugriff auf Typebene festgelegt werden, indem die Instanzen einzelner Typen ausgeblendet oder als nur lesbar definiert werden (vgl. Abschnitt 2.5).

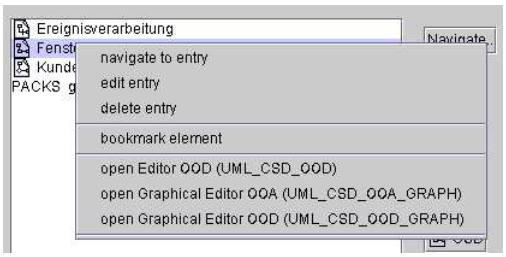
Rollenspezifische Sichten

Für die Zuordnung geeigneter Sichten zu einer bestimmten Rolle stellt das OMS keinen Mechanismus zur Verfügung. Er mußte daher in *genform* implementiert werden und stellt sicher, daß ein Benutzer mit einer bestimmten Rolle nur bestimmte Werkzeugsichten erhält. Dadurch wird die Arbeitsumgebung des Benutzers an seine Rolle angepaßt.

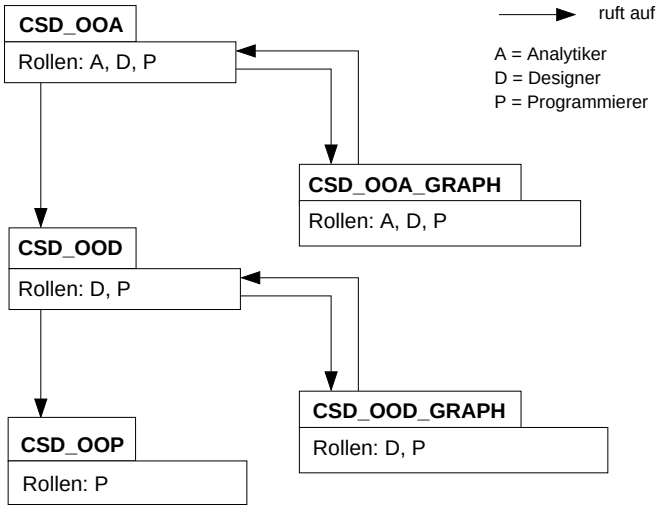
Abbildung 4.20(a) zeigt das Datenmodell für die Zuordnung von Rollen zu Werkzeugschemata: Jedem WZS kann eine Menge von Rollen zugeordnet werden – sind keine zugeordnet, können alle Rollen auf das WZS zugreifen. Abbildung 4.20(c) zeigt verschiedene WZS mit ihren Navigationsbeziehungen. Bei jedem WZS sind die Rollen angegeben, die auf das WZS zugreifen können. In Abbildung 4.20(b) ist ein Menü dargestellt, mit dem Werkzeuge auf einem Klassendiagramm aufgerufen werden. Das Menü enthält Analyse- und Entwurfswerkzeuge, entspricht also der Sicht des Designers. Jeder Menüeintrag enthält in Klammern den



(a) Datenmodell



(b) Menü mit Werkzeugen/WZS in der Sicht eines Designers



(c) Navigationsbeziehungen zwischen WZS und zugeordnete Rollen

Abbildung 4.20: Zuordnung von Werkzeugschemata zu Rollen

Namen des Werkzeugschemas, in dem das Werkzeug definiert ist. Beim Erzeugen des Menüs wird geprüft, ob dem Zielschema Rollen zugeordnet sind. Trifft dies zu, wird der Menüeintrag nur eingeblendet, wenn die aktuelle Rolle des Benutzers in der Menge von Rollen enthalten ist, die dem Schema zugeordnet ist.

4.3.2 Planung und Management

Planungs- und Management-Tätigkeiten haben das Ziel, den Ablauf eines konkreten Projekts unter Berücksichtigung der Projektziele, der verfügbaren Ressourcen, von Vorgaben für Zeit und Kosten und weiterer Randbedingungen festzulegen. Grundlage dafür ist das Prozeßmodell, das Vorgaben für die Art der Aufgaben und ihrer Beziehungen macht.

Planung und Management werden hier unterschieden, weil sie im Lebenszyklus des Projekts als aufeinanderfolgende Phasen erscheinen (vgl. Abbildung 4.17).

Planung

In der Planungsphase wird der Projektverlauf im vorhinein festgelegt. Die Planung liefert

- die hierarchische Zerlegung des Projekts in einzelne überschaubare Aufgaben, die von einzelnen Personen oder kleinen Gruppen ausgeführt und deren Ausgänge geprüft werden können. *Meilensteine* markieren besondere, meßbare Ereignisse wie die Fertigstellung eines Teilsystems oder den Abschluß einer Entwicklungsphase.
- eine Schätzung der Aufwände einzelner Aufgaben und die resultierenden Anfangs- und Endtermine unter Berücksichtigung von Abhängigkeiten zwischen Aufgaben und dem Ressourcenbedarf. Der Zeitaufwand bestimmt auch wesentlich die zu erwartenden Kosten.
- den geschätzten Ressourcenbedarf. In Software-Projekten sind hier die Personen relevant, die die Aufgaben ausführen. Bei der Verplanung von Personen müssen Randbedingungen wie ihre Verfügbarkeit sowie Kenntnisse und Fähigkeiten berücksichtigt werden.
- eine erste Zuordnung von Software-Dokumenten zu Aufgaben. Bei der Wartung eines Softwaresystems ergibt sie sich aus den zu erwartenden Änderungen an vorhandenen Dokumenten; bei einer Neuentwicklung aus der geplanten Systemstruktur (Teilsysteme, Module, Pakete) und dem gewählten Prozeßmodell mit seinen erforderlichen Dokumenten.

Das Prozeßmodell definiert Konsistenzbedingungen, denen der Plan genügen muß: *Statische Konsistenzbedingungen* müssen immer erfüllt sein, eine Verletzung ist nicht möglich. So können keine Aufgaben oder Beziehungen zwischen Aufgaben erzeugt werden, die im Prozeßmodell nicht vorgesehen sind. Hierfür sorgt das OMS, das den Plan verwaltet und dabei die Konsistenzbedingungen aus dem Datenbankschema berücksichtigt. *Dynamische Konsistenzbedingungen* werden auf Anforderung geprüft, es können also inkonsistente Zustände in einem Plan entstehen. Kardinalitäten an Aufgabentypen, Beziehungen und Dokumentordnern müssen mit Prüfwerkzeugen geprüft werden, da beim Bearbeiten des Plans die Konsistenzbedingungen nicht ständig eingehalten werden können.

Operationen auf dem Aufgabennetz: Der Planer kann folgende Operationen auf dem Aufgabennetz durchführen:

- Erzeugen und Löschen von Aufgaben
- Bearbeiten der Eigenschaften von Aufgaben (Name, Sollaufwände, Solltermine)
- Erzeugen und Löschen von Nachfolgerbeziehungen zwischen Aufgaben
- Erzeugen und Löschen von Beziehungen zwischen Aufgaben und Benutzern (Zuordnen des Bearbeiters zu einer Aufgabe)
- Erzeugen und Löschen vom Beziehungen zwischen Ordnern und Dokumenten (Zuordnen von Dokumenten zu einer Aufgabe)
- Berechnen von Sollterminen unter Berücksichtigung der Abhängigkeiten zwischen Aufgaben (Netzplanberechnung)

Darstellung des Projektplans: Zur Visualisierung des Projektplans benötigt der Planer folgende Darstellungen:

- Baum- oder Tabellendarstellung der Aufgabenzerlegung
- Netzplan mit Aufgaben und ihren Abhängigkeiten, sowie Sollaufwänden und Sollterminen
- Kalenderdarstellung des Projekts (Gantt-Darstellung)
- Darstellung der Ressourcenauslastung

Management

Die Planung wird vor dem Beginn der Entwicklungstätigkeiten durchgeführt – Management-Tätigkeiten werden parallel zum laufenden Projekt durchgeführt. Der Manager geht von dem Projektplan aus, den die Planung geliefert hat, und paßt ihn an die aktuelle Projektsituation an. Diese Anpassung und Erweiterung des Projektplans ist aus zwei Gründen nötig:

- Der ursprüngliche Plan ist in den meisten Fällen nicht vollständig, da viele Entscheidungen über das weitere Vorgehen erst getroffen werden können, wenn ein Teil der Arbeit bereits getan ist [119].
- Es muß auf die aktuelle Projektsituation reagiert werden: Geänderte Anforderungen, technische Probleme oder Ressourcenknappheit können eine teilweise Neuplanung des Projekts erfordern.

Wichtig für den Manager ist es also, Informationen über den Projektverlauf zu erhalten. Zum einen auf Anforderung, indem er sich wichtige Informationen ansieht oder durch Anfrage ermittelt, zum anderen durch automatische Benachrichtigung über wichtige Ereignisse.

Zur Beeinflussung des weiteren Ablaufs stehen dem Manager die gleichen Operationen zur Manipulation des Prozeßzustands zur Verfügung wie dem Planer – allerdings sollten die Möglichkeiten zum Bearbeiten des Aufgabennetzes abhängig vom Zustand der einzelnen Aufgaben eingeschränkt werden [158]. Zusätzlich stehen ihm folgende Operationen zur Verfügung:

- Bearbeiten des Aufgabenzustands: Mögliche Zustände und Zustandsübergänge werden durch die Zustandsdiagramme von Aufgaben festgelegt. Ein Zustandsübergang kann also nur ausgelöst werden, wenn seine Bedingung erfüllt ist. Allerdings ist es denkbar, dem Manager auch 'ungültige' Zustandswechsel zu gestatten. Diese können nötig werden, wenn aufgrund widriger Umstände vom geplanten Vorgehen abgewichen werden muß. So könnte es nötig sein, ein Dokument an eine Folgeaufgabe weiterzuleiten, obwohl es noch unvollständig oder nicht geprüft ist.
- Eintragen von tatsächlichen Aufwänden und Terminen: Das Ende der Bearbeitung kann auch automatisch erfaßt werden, wenn eine Aufgabe abgeschlossen wird. Auch für die Aufwandserfassung ist eine Werkzeugunterstützung denkbar.

4.3.3 Entwicklung

Bei der Ausführung des Software-Prozesses muß zwischen Entwicklungs- und Management-Tätigkeiten unterschieden werden. Erstere umfassen alle technischen Aufgaben zur Entwicklung oder Wartung eines Softwaresystems. Die Entwicklungstätigkeiten werden von

Management-Tätigkeiten begleitet. Sie sorgen dafür, daß alle Entwicklungsaufgaben zum erforderlichen Zeitpunkt und korrekt durchgeführt werden können, und zum Erreichen des Projektziels beitragen.

Für den PSEU-Entwickler sind hier zwei Fragen interessant: Zum einen die Frage, welche Dokumente mit welchen Werkzeugen bearbeitet werden. Die PSEU muß diese Werkzeuge passend für eine gegebene Aufgabe und Rolle anbieten. Zum anderen die Frage, welche Sicht die Benutzer auf den Prozeß haben. Auch hier müssen passende Werkzeuge zur Visualisierung und Beeinflussung des Prozeßzustands angeboten werden.

Prozeßsicht. Den Entwickler interessieren primär die von ihm zu erledigenden Aufgaben. Über eine Aufgabenliste (*Agenda*) kann er auf folgende Informationen zugreifen:

– Die ihm zugeordneten Aufgaben und deren Zustand. Die möglichen Zustände der Beziehung zwischen einer Aufgabe und ihrem Bearbeiter sind im Zustandsdiagramm in Abbildung 4.21 skizziert.

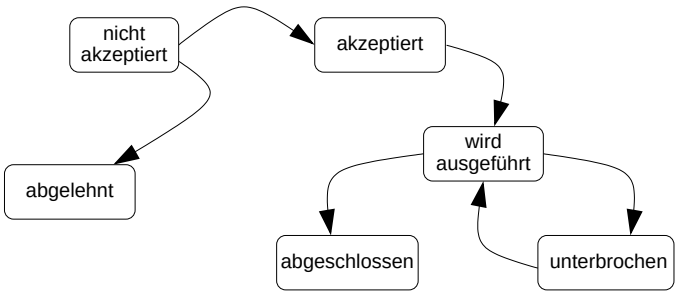


Abbildung 4.21: Zustandsdiagramm für die Beziehung zwischen Entwickler und Aufgabe

– Eine Beschreibung jeder Aufgabe, ihrer Ziele und Randbedingungen. Sie dient dem Entwickler als Erläuterung und Anleitung.
– Aufwände und Termine. Der Planer legt Sollaufwand und Solltermine für Beginn und Ende einer Aufgabe fest, die dem Entwickler als Orientierung dienen können; der Entwickler trägt die tatsächlichen Aufwände ein oder die Aufwände werden automatisch erfaßt. Die tatsächlichen Termine ergeben sich automatisch beim Bearbeiten der Aufgabe.
– Die relevanten Dokumente. Sie befinden sich in den Dokumentordnern der Aufgaben.
– Die zugehörigen Werkzeuge. Sie sind über die Dokumentordnern erreichbar und dienen zum Betrachten oder zum Bearbeiten der Dokumente im Ordner.

Neben seinen eigenen sind für den Entwickler auch Aufgaben in der 'Umgebung' seiner Tätigkeit interessant: Vorgänger-Aufgaben, da von ihnen seine Aufgaben abhängen; Nachfolger-Aufgaben, da sie das weitere Vorgehen im Prozeß bestimmen. Das Aufgabennetz muß für den Entwickler möglichst übersichtlich dargestellt werden und den Zugriff auf die Zustände und Bearbeiter von Aufgaben ermöglichen – letzteres, um die Kommunikation im Prozeß zu erleichtern.

Dokumente und Werkzeuge. Die Entwicklungstätigkeiten haben das Ziel, Software-Dokumente zu erzeugen, zu bearbeiten, zu prüfen oder zu transformieren. Die CASE-Werkzeuge der Entwickler müssen an die Anforderungen der konkreten Aufgaben und Rollen angepaßt sein. Diese Anpassung kann nur erreicht werden, wenn eine enge Beziehung zwischen Prozeßmodell, Prozeßzustand und Werkzeugen besteht [101]:

1. Durch die Zuordnung von Werkzeugschemata zu Dokumentordnern stehen dem Entwickler die passenden Werkzeuge zur Anzeige und Bearbeitung von Dokumenten zur Verfügung – genauer handelt es sich um passende *Werkzeugsichten*. Eine Werkzeugsicht faßt zusammen: Eine Menge von sichtbaren und manipulierbaren Konzepten, eine Menge von Darstellungen und eine Menge von Kommandos. Letztere umfassen sowohl Editierkommandos zum Erzeugen, Löschen und Bearbeiten von Einträgen, als auch dokumentspezifische Kommandos etwa zum Prüfen oder Transformieren eines Dokuments.

2. Durch Zugriffsrechte an Dokumenten werden die Möglichkeiten zur Anzeige und Manipulation einzelner Einträge eingeschränkt. Zugriffsrechte gelten für einzelne Benutzer oder Rollen und hängen vom aktuellen Prozeßzustand ab. In herkömmlichen Werkzeugen sind Zugriffsrechte nur sehr eingeschränkt nutzbar: Es stehen nur sehr wenige Rechtemodi zur Verfügung (lesen, schreiben) und Rechte können nur sehr grobgranular vergeben werden (auf ganzen Dokumenten).

   In *genform*-Dokumenten können Zugriffsrechte hingegen auf allen Dokumenteinträgen vergeben werden, so daß der Zugriff auf beliebig feine Granulate geregelt werden kann. PCTE bietet eine große Zahl von Rechtemodi an, mit denen exakt festgelegt wird, welche Änderungen erlaubt sind (vgl. Abschnitt 2.5.3). Die *genform*-Werkzeuge berücksichtigen die Zugriffsrechte, passen also die verfügbaren Darstellungen und Kommandos an die Rechtesituation an. Somit wird durch die Rechtesituation das Verhalten der Werkzeuge für einen bestimmten Benutzer in einer bestimmten Prozeßsituation beeinflußt. Die Zugriffsrechte werden aber nicht vom Werkzeug, sondern von der Prozeßmaschine festgelegt; somit bleiben die Werkzeuge unabhängig vom Prozeßmodell.

3. Entscheidend für den Prozeßverlauf sind die Zustände von Dokumenten. Sie bestimmen, ob die Bearbeitung eines Dokuments abgeschlossen ist und ob das Dokument an eine Folgeaufgabe weitergeleitet werden kann. Im ersten Fall kann der Zustand einer Aufgabe auch direkt aus den Zuständen der bearbeiteten Dokumente abgeleitet werden.

   Ein Dokument hat *implizite Zustände*: Es kann fehlerfrei sein, konsistent mit dem Analysedokument, geändert oder noch nicht übersetzt – der implizite Zustand spiegelt alle Eigenschaften des Dokuments wider, die in einer bestimmten Situation relevant sind. Ein Teil dieser Zustände kann *explizit* gemacht und in einem Zustandsdiagramm beschrieben werden. Zur Steuerung des Prozesses können nur die expliziten Zustände berücksichtigt werden. Implizite Zustände müssen also zunächst explizit gemacht werden, mit einem Analysewerkzeug, das die Dokumenteigenschaften prüft, oder durch den Benutzer, der das Dokument beurteilt und manuell einen Zustand auswählt.

   Welche Eigenschaften eines Dokuments relevant sind, und damit welche expliziten Zustände es besitzt, hängt von der Aufgabe ab, in der das Dokument bearbeitet wird. In *genform* werden die Zustandsdiagramme den Dokumentordnern einer Aufgabe zugeordnet. Dokumentzustände werden also nicht im Werkzeug oder Datenmodell des Dokuments definiert, sondern im Prozeßmodell. Der Entwickler muß nun die Möglichkeit haben, den Dokumentzustand zu beeinflussen – möglichst direkt mit seinem CASE-Werkzeug.

Mit *genform* gebaute CASE-Werkzeuge bieten die Möglichkeit an, den Dokumentzustand wie ein Attribut des Dokuments zu ändern. Dabei wird das Zustandsdiagramm berücksichtigt, es können also nur erlaubte Zustandswechsel ausgeführt werden. Die Integration von Dokumentverwaltung und Prozeßsteuerung wird erreicht, indem im Datenmodell der Dokumente Verweise zwischen Dokumenten und Dokumentzuständen vorgesehen werden. Entscheidend ist, daß in den Werkzeugen keine Annahmen über die Verwaltung der Zustände (oder allgemein: prozeßspezifischer Eigenschaften) enthalten sind, da sonst die Wartbarkeit und Wiederverwendbarkeit der Werkzeuge erschwert würden.

Mehr zu den Prozeßwerkzeugen in Abschnitt 5.2; Abschnitt 5.2.4 enthält ein Beispiel für die Realisierung eines Zustandsautomaten für Dokumente und zeigt ein Werkzeugfenster, mit dem der Zustand manipuliert werden kann (Abbildung 5.11 auf Seite 185).

4.3.4 Striktheit der Prozeßunterstützung

Die Striktheit der Prozeßunterstützung bestimmt, wie stark die Aktionen der Benutzer einer PSEU eingeschränkt werden (vgl. Seite 137). Beim Festlegen der Striktheit müssen folgende Faktoren berücksichtigt werden:

- Zu starke Einschränkungen können zu psychologischen Problemen führen [11], die die Arbeit der Entwickler beeinträchtigen. Schließlich handelt es sich meist um gut ausgebildete Spezialisten, die eigene Vorstellungen von einem korrekten Vorgehen haben [312].
- Andererseits können Abweichungen vom vorgegebenen Prozeßmodell negative Auswirkungen auf den Fortschritt und die Ergebnisse eines Projekts haben, etwa, wenn Qualitätssicherungsmaßnahmen ausgelassen, oder Aufgaben mehrfach ausgeführt werden, oder wenn von falschen Dokumenten ausgegangen wird. Schließlich bietet das Prozeßmodell auch eine Basis für die Kollaboration und die Kommunikation der Entwickler [20].
- Eine strikte Prozeßunterstützung ist natürlich nur so lange sinnvoll, wie der tatsächliche Prozeß auch gemäß Prozeßmodell durchgeführt werden kann. Ist dies nicht möglich, weil bestimmte Ereignisse, Situationen und Ausnahmen im Modell nicht berücksichtigt wurden, wird die Prozeßunterstützung zum Prozeßhindernis.

Eine PSEU sollte also auch bezogen auf die Striktheit der Prozeßunterstützung flexibel sein. In *genform*-PSEU wird die Prozeßinstanz durch ein Aufgabennetz repräsentiert. Die Striktheit der Prozeßunterstützung wird durch das Prozeßmodell bestimmt, also durch die Beziehungen zwischen Aufgabentypen und ihren Kardinalitäten, durch vorgegebene Datenflußbeziehungen und Dokumentzustände.

- *Passive guidance* wird erreicht, wenn Planer, Manager und Entwickler ihre Aufgaben selbst erzeugen, also das Aufgabennetz unter Berücksichtigung des Prozeßschemas erweitern. Wie stark die Vorgaben für das Erzeugen und Verknüpfen von Aufgaben sind, hängt vom Umfang und der Detaillierung des Prozeßmodells ab, liegt also im Ermessen des Prozeßmodellierers.
- *Active guidance* wird erreicht, wenn die Prozeßmaschine Aufgaben selbständig erzeugt. Dies ist in der aktuellen Implementierung nur in einfachen Fällen vorgesehen – wenn das Prozeßmodell die Existenz einer minimalen Anzahl von Aufgaben eines Typs vorschreibt.

Die Prozeßmaschine könnte allerdings so erweitert werden, daß komplexere Mechanismen den Prozeß planen und entscheiden, welche Aufgaben auszuführen sind. Den Aufgaben könnte auch automatisch ein Bearbeiter zugeordnet werden, sofern ausreichende Informationen (etwa über die Auslastung der Mitarbeiter und Fälligkeitstermine) zur Verfügung stehen und ein passender Planungsmechanismus implementiert würde. Mehr zum Aufbau der Prozeßmaschine in Abschnitt 5.1.1. Eine wichtige Rolle bei der Realisierung von *active guidance* spielt das Repository, das die Prozeßmaschine über relevante Änderungen des Prozeßzustands informiert.

– *Process enforcement* wird erreicht, indem der Zugriff auf Dokumente und Werkzeuge eingeschränkt wird. Zur Regelung des Zugriffs auf Dokumente werden Zugriffsrechte vergeben. Diese gelten für alle Werkzeuge, die auf der Objektbank arbeiten, könnten aber mit einem Administrationswerkzeug und passenden Rechten verändert werden. Die Benutzer der PSEU greifen über Dokumentordner auf die Dokumente zu. Die Dokumentordner dienen weniger dem Zugriffsschutz, als der Vereinfachung des Zugriffs, da der Benutzer im Ordner die für seine Aufgabe relevanten Dokumente vorfindet. Er kann alternativ auch über die Dokumentstruktur navigieren, wo wiederum die Zugriffsrechte berücksichtigt werden. Gleiches gilt für die Werkzeuge, die den Ordnern zugeordnet sind: Sie stehen dem Benutzer direkt für die Ausführung der Aufgabe zur Verfügung und sind an die konkreten Anforderungen der Aufgabe (etwa Notationen und Kommandos) angepaßt. Im Werkzeug werden die sichtbaren Daten und die ausführbaren Kommandos durch die Zugriffsrechte am Dokument gesteuert, also gleichzeitig unerlaubte Zugriffe verhindert und die Bedienung des Werkzeugs vereinfacht. Voraussetzung für die Steuerung der Werkzeuge über Zugriffsrechte sind feingranulare Dokument-Datenmodelle und feingranular vergebene Zugriffsrechte.
– *Process automation* wird mit Komponenten erreicht, die die Prozeßmaschine erweitern und Aktionen automatisch ausführen. *genform* liefert hierzu lediglich die Infrastruktur und Schnittstellen, über die spezielle Komponenten eingebunden werden können (Abschnitt 5.1.1).

4.3.5 Andere Ansätze

Ziel von *genform* ist es, Planung und Management in PSEU zu unterstützen, nicht zu automatisieren. Ähnlich geht auch *AHEAD* [179] vor, das ein *"human-centered environment"* zum Bearbeiten von Plänen und zur Steuerung von Prozessen zur Verfügung stellt. Aus Benutzersicht ähneln die Aufgabennetze in *genform* denen in *DYNAMITE* [157]. Die Unterschiede liegen allerdings in der Modellierung (Datenbank-Schemata vs. *PROGRES*-Regeln [291]), der Verwaltung (OMS vs. Graphenspeicher *GRAS* [211]) und der Ausführung (komponentenbasierte Prozeßmaschine vs. *PROGRES*-Interpreter).

In regelbasierten Ansätzen wie *Marvel* [126] wird auch die Planung automatisiert, indem die nächsten auszuführenden Schritte automatisch bestimmt werden. Sie eignen sich aber nur für sehr technische Prozesse wie Konfigurationsverwaltung und Systemgenerierung, da Software-Prozesse zu komplex, änderungsanfällig und kreativ sind, um einer starren Automatisierung zugänglich zu sein [75, 123]. Weiteres Problem: Die automatischen Planer sind sehr komplex und ihre Entscheidungen für die (menschlichen) Projektbeteiligten oft nur schwer nachvollziehbar.

Auch in *OIKOS* [251] werden die Eigenschaften der verfügbaren Werkzeuge explizit beschrieben und ein Arbeitskontext bereitgestellt: Ein *tool repository* speichert die Werkzeugeigenschaften wie die 'Signatur' des Werkzeugs (Typen der Ein- und Ausgabedokumente), und die Parameter, die beim Werkzeugaufruf übergeben werden müssen. Über die Zuordnung von Werkzeugen zu Dokumenttypen kann entschieden werden, welche Werkzeuge auf einem gegebenen Dokument genutzt werden; umgekehrt wird dafür gesorgt, daß alle benötigten Dokumente bereitstehen. Werkzeuge werden über *envelopes* in die Umgebung integriert. Dabei geht *OIKOS* von vorhandenen Werkzeugen aus – die Anpassung der Werkzeuge ist also nur sehr eingeschränkt möglich. In *genform* werden die Werkzeugeigenschaften hingegen abhängig von der Aufgabe spezifiziert und die Werkzeuge passend gebaut. Auch können in *genform* wegen des feingranularen Datenmodells Werkzeuge für beliebige Granulate definiert werden.

Der komponentenbasierte Ansatz von *Endeavors* [37] geht weiter als der von *genform*: *Endeavors* nutzt Internet-Technologien für die Verteilung von Komponenten und die Integration verteilter Werkzeuge. *genform* geht hingegen von einem zentralen Repository aus und berücksichtigt die Verteilung der Werkzeugkomponenten nicht explizit. Internet-Protokolle werden lediglich für die Kommunikation zwischen Werkzeugen und OMS genutzt.

In *ALF* [47] modelliert ein *MASP* alle Aspekte einer einzelnen Tätigkeit eines einzelnen Benutzers, darunter auch die zu bearbeitenden Daten in Form von PCTE-SDS. Typdefinitionen werden zwischen den SDS (und damit zwischen den zugehörigen MASP) importiert und erweitert. Die Definition des Prozeßmodells ist also wie in *genform* eng mit Datenbankschemata und Sichten verknüpft. Bei einer Instantiierung eines MASP wird ein Arbeitsbereich eingerichtet und mit den zu bearbeitenden Objekten (Instanzen der Typen aus dem Datenmodell des MASP) gefüllt. Auch in *ALF* werden alle Prozeßdaten persistent in PCTE verwaltet. Allerdings sind die Produktdaten hier grobgranular modelliert. Änderungen im Prozeß werden auf Änderungen an den Daten in der Objektbank abgebildet. Die MASP-Instanz enthält Trigger, die auf Benutzerereignisse oder das Ändern von Daten reagieren.

Kapitel 5

Realisierung von PSEU

In diesem Kapitel wird beschrieben, wie eine Prozeßmaschine mit *genform* gebaut und mit den Werkzeugen der PSEU integriert werden kann: Abschnitt 5.1 erläutert die Zusammensetzung der Prozeßmaschine aus Komponenten, den Prozeßagenten. Abschnitt 5.2 geht auf die Werkzeuge ein: Zum einen auf die Prozeßwerkzeuge, die aus dem Prozeßmodell generiert werden; zum anderen auf die CASE-Werkzeuge, die in die PSEU integriert werden müssen. Abschnitt 5.2.4 enthält schließlich ein ausführliches Beispiel zur Verwaltung von Dokumentzuständen. Abschnitt 5.3 geht auf andere Ansätze zur Realisierung von PSEU ein; Abschnitt 5.4 stellt Meta-Werkzeuge vor, die Werkzeugentwickler und Prozeßmodellierer bei ihrer Arbeit unterstützen.

5.1 Prozeßmaschine

Die *Prozeßmaschine* (*Process Engine*, PE) ist die reaktive Komponente einer PSEU: Sie reagiert auf Änderungen im Prozeß und steuert den weiteren Prozeßfortschritt. Änderungen des Prozeßzustands resultieren aus

- Änderungen der Zustände oder der Inhalte von Dokumenten. Dokumente können unter ganz unterschiedlichen Gesichtspunkten charakterisiert werden, etwa nach dem Bearbeitungszustand (neu, geändert) oder Qualitätskriterien (fehlerfrei, fehlerhaft). Um die Eigenschaften besser handhaben zu können, werden sie auf explizite Dokumentzustände abgebildet. Diese Abbildung können spezielle Prüfroutinen leisten, oder der Benutzer, etwa am Ende einer Review-Sitzung.
- Änderungen der Zustände, Eigenschaften oder Beziehungen von Aufgaben. Zustandsänderungen können manuell vom Benutzer ausgelöst werden, etwa wenn er eine Aufgabe als abgeschlossen betrachtet. Zustandsänderungen können auch automatisch ausgelöst werden, etwa wenn sich der Zustand eines zugeordneten Dokuments ändert. Wichtige Eigenschaften von Aufgaben sind Aufwände und Termine, die bei der Prozeßdurchführung berücksichtigt werden müssen. Außerdem sind die Kontrollflußbeziehungen zwischen Aufgaben und die Beziehungen zu den ausführenden Benutzern relevant: Eine Aufgabe kann erst aktiv sein, wenn alle Vorgänger abgeschlossen sind und ein Bearbeiter zugeordnet ist.

Die Zustände von Aufgaben und Dokumenten werden in Zustandsdiagrammen beschrieben (vgl. Abschnitt 4.2.5 auf Seite 152). Die Prozeßmaschine muß die zugehörigen Zustandsautomaten verwalten, also die Bedingungen an den Zustandsübergängen überwachen und beim Zustandswechsel die zugeordneten Aktionen ausführen. Weiterhin muß sie

- Aufgaben und Beziehungen zwischen Aufgaben erzeugen, etwa die Unteraufgaben einer komplexen Aufgabe oder obligatorische Nachfolger.
- Dokumente verwalten und über Datenflüsse an Folgeaufgaben weiterleiten.

Um den Aufwand für den Bau von PSEU zu reduzieren, sollten möglichst viele *genform*-Komponenten (vgl. Abschnitt 3.3.2 auf Seite 114) in der Prozeßmaschine wiederverwendet werden. Außerdem wird durch die Wiederverwendung von Komponenten für den OMS-Zugriff die Integration von Werkzeugen und Prozeßmaschine erleichtert und dafür gesorgt, daß sich die Prozeßmaschine beim OMS-Zugriff ähnlich wie die Werkzeuge verhält – etwa bei der Behandlung von Fehlern.

GUI-Komponenten werden für die Benutzungsschnittstelle von Prozeßwerkzeugen verwendet. Diese Werkzeuge sind zum Teil universell, zum Teil prozeßspezifisch, wenn sie Aufgabentypen und Beziehungen des Prozeßmodells berücksichtigen. Auch Prozeßwerkzeuge werden mit Werkzeugschemata und Werkzeugkomponenten gebaut: Die Werkzeugschemata werden aus dem Prozeßmodell generiert, da das Prozeßmodell die Eigenschaften der Werkzeuge bestimmt. Neben den vorhandenen *genform*-Komponenten werden einige spezielle benötigt, etwa zur graphischen Anzeige von Aufgaben im Prozeßeditor. Ziel ist natürlich, mit möglichst wenigen Erweiterungen auszukommen. Spezielle Komponenten werden auch in der Prozeßmaschine zur Verwaltung und Manipulation des Prozeßzustands benötigt. Die Bausteine der Prozeßmaschine heißen *Prozeßagenten*: Sie erlauben die Anpassung der PE an die Anforderungen des jeweiligen Prozesses.

5.1.1 Prozeßagenten

Vergleicht man die Beschreibung eines Software-Prozesses mit der eines Informationssystems, so kann man auch beim Software-Prozeß die Spezifikation der Architektur und die eigentliche Implementierung unterscheiden [299]: Graphische Prozeßmodelle stellen die *Architektur* eines Prozesses dar. Die PE kann den Kontroll- und Datenfluß anhand des Prozeßmodells steuern; die Regeln, wie diese Steuerung funktioniert, stecken in der PE. In der *Implementierung* des Prozesses wird auf niedrigerer Abstraktionsebene und mit höherem Detaillierungsgrad beschrieben, welche konkreten Aktionen aufgrund von Änderungen des Prozeßzustands ausgeführt werden.

Die PE ist aus Komponenten aufgebaut, den *Prozeßagenten*. Die Prozeßagenten implementieren die Funktionen der PE; ihre Zusammensetzung wird durch eine Architekturspezifikation bestimmt. Die Architektur und Implementierung der PE steht also in direktem Zusammenhang mit der Architektur und Implementierung des Prozesses. Mit den Prozeßagenten kann die PE flexibel an die Anforderungen des jeweiligen Prozesses angepaßt werden. Prozeßagenten sind keine 'vollwertigen' Agenten etwa im Sinne von [186]. Sie sind allerdings aktive Entitäten, die den Prozeß überwachen und selbständig Aktionen ausführen.

Grundlage für die Arbeit der Prozeßagenten sind die im Prozeßmodell definierten Aufgaben und ihre Beziehungen. Es handelt sich also um einen aufgabenbasierten Ansatz (vgl. Punkt 1 auf Seite 56): Jeder Aufgabe ist ein *Task Execution Agent* (TEA) zugeordnet, der für die Ausführung der Aufgabe verantwortlich ist. Er kennt den Kontext der Aufgabe (Vorgänger, Nachfolger, Unteraufgaben, über Datenflüsse verbundene Aufgaben) und das zugrundeliegende Prozeßmodell.

Agentenhierarchie

Der TEA lädt weitere Agenten, die verschiedene Funktionen implementieren. Jeder Agent kann weitere Agenten laden, so daß sich eine baumartige *Agentenhierarchie* ergibt, deren Wurzel der TEA ist. Über die Beziehungen zwischen Agenten kann ein übergeordneter Agent die Dienste der untergeordneten Agenten aufrufen und können die untergeordneten Agenten Ergebnisse zurückliefern (Abbildung 5.1).

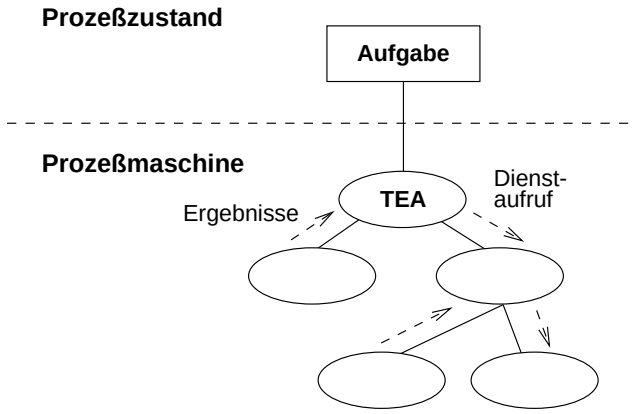


Abbildung 5.1: Kommunikation zwischen Agenten

Welche Subagenten ein Agent lädt, wird festgelegt durch das Prozeßmodell, in dem die Subagenten spezifiziert werden, oder durch die Implementierung des Agenten selbst. Im ersten Fall kann also die Agentenhierarchie und damit der Aufbau der PE durch das Prozeßmodell gesteuert werden – ähnlich wie die Eigenschaften von CASE-Werkzeugen durch Werkzeugschemata gesteuert werden (Abschnitt 3.2 auf Seite 105).

Werkzeugschemata für die Prozeßmaschine

Tatsächlich wird hier der gleiche Mechanismus verwendet: Auch die Eigenschaften der PE werden über Werkzeugschemata spezifiziert. Allerdings müssen diese Werkzeugschemata nicht vollständig vom Prozeßmodellierer spezifiziert werden, sondern werden aus dem Prozeßmodell generiert. Das Werkzeugschema für die PE enthält

- das Datenmodell zur Verwaltung des Prozeßzustands. Es besteht aus Aufgabentypen mit ihren Beziehungen und den Typen der Dokumentordner.
- Werkzeugparameter zur Spezifizierung der Eigenschaften von Prozeßelementen. Hierzu zählen die Prozeßagenten mit ihren Parametern sowie die Eigenschaften von Dokumentordnern und Datenflußbeziehungen.

Die PE funktioniert also ähnlich wie die CASE-Werkzeuge: Sie wird zur Laufzeit aus generischen Komponenten zusammengesetzt, die das Datenmodell und die Werkzeugparameter interpretieren.

Framework-Komponenten

In Abbildung 5.2 ist ein Ausschnitt aus der Klassenhierarchie für Prozeßagenten dargestellt. Die Wurzel der Klassenhierarchie ist ProcessAgent. Die Schnittstelle enthält Operationen zum Initialisieren, zum Aktivieren und Deaktivieren des Agenten, sowie zum Setzen der Werkzeugschemata. Letztere müssen gesetzt werden, wenn der Agent auf dokumentspezifische Informationen zugreifen soll, also das Datenmodell der Dokumente kennen muß. In diesem Fall benötigt der Agent auch eine eigene Sicht auf die Objektbank: Hierzu wird ein H-PCTE-Subprozeß für den Agenten gestartet und das Arbeitsschema passend gesetzt. Alle anderen Prozeßagenten werden im H-PCTE-Prozeß der PE ausgeführt, in dessen Sicht der Prozeßzustand manipuliert werden kann.

Dank des Transaktionsmechanismus von H-PCTE [270] können die Subprozesse innerhalb einer PE-Ausführung kooperieren, so daß keine Sperrkonflikte auftreten. Konflikte sind aber zwischen verschiedenen PSEU-Ausführungen möglich – die Arbeit der verschiedenen Benutzer und PSEU muß daher durch das Prozeßmodell so geregelt werden, daß möglichst wenige Konflikte auftreten.

Ein ComplexAgent kann andere Agenten laden und verwalten. Beispiele für komplexe Agenten sind FolderManager und DocumentManager. In Abbildung 5.3 sind die Instanzen der Framework-Komponenten dargestellt, wie sie als Agentenhierarchie einer Aufgabe zugeordnet sind: Die Wurzel der Hierarchie bildet der TaskAgentEnvelope. Er verwaltet genau einen Agenten und aktiviert ihn abhängig davon, ob die Aufgabe aktiv ist – er dient also als Adapter zwischen dem persistenten Prozeßzustand und den Agenten. Hierzu überwacht er das Zustandsattribut am Aufgabenobjekt in der Objektbank.

In der Agentenhierarchie aus Abbildung 5.3 ist der Aufgabe A ein Agent vom Typ TaskExecutionAgent zugeordnet – es könnte jedoch auch ein anderer Agent im Prozeßmodell spezifiziert werden. Der TaskExecutionAgent lädt weitere Agenten. Welche dies sind, wird im Prozeßmodell spezifiziert, ebenso die Parameter.

TaskManager: Der TaskManager überwacht das Aufgabennetz und erzeugt automatisch Unteraufgaben von A. Dazu ermittelt er die Typen und Eigenschaften der Beziehungen zu Subaufgaben aus dem Werkzeugschema. Damit eine Subaufgabe automatisch erzeugt werden kann, muß eine der folgenden Bedingungen gelten:

- Die Anzahl der Instanzen eines Aufgabentyps ist kleiner als die untere Grenze der Kardinalität.
- Die Anzahl der ausgehenden Beziehungen zu Teilaufgaben ist kleiner als die untere Grenze der Kardinalität und der Beziehungstyp hat nur einen Zielaufgabentyp.

Das automatische Erzeugen von Aufgaben kann über Parameter abgeschaltet werden. Treffen die Bedingungen nicht zu, müssen die Unteraufgaben vom Manager des Prozesses manuell

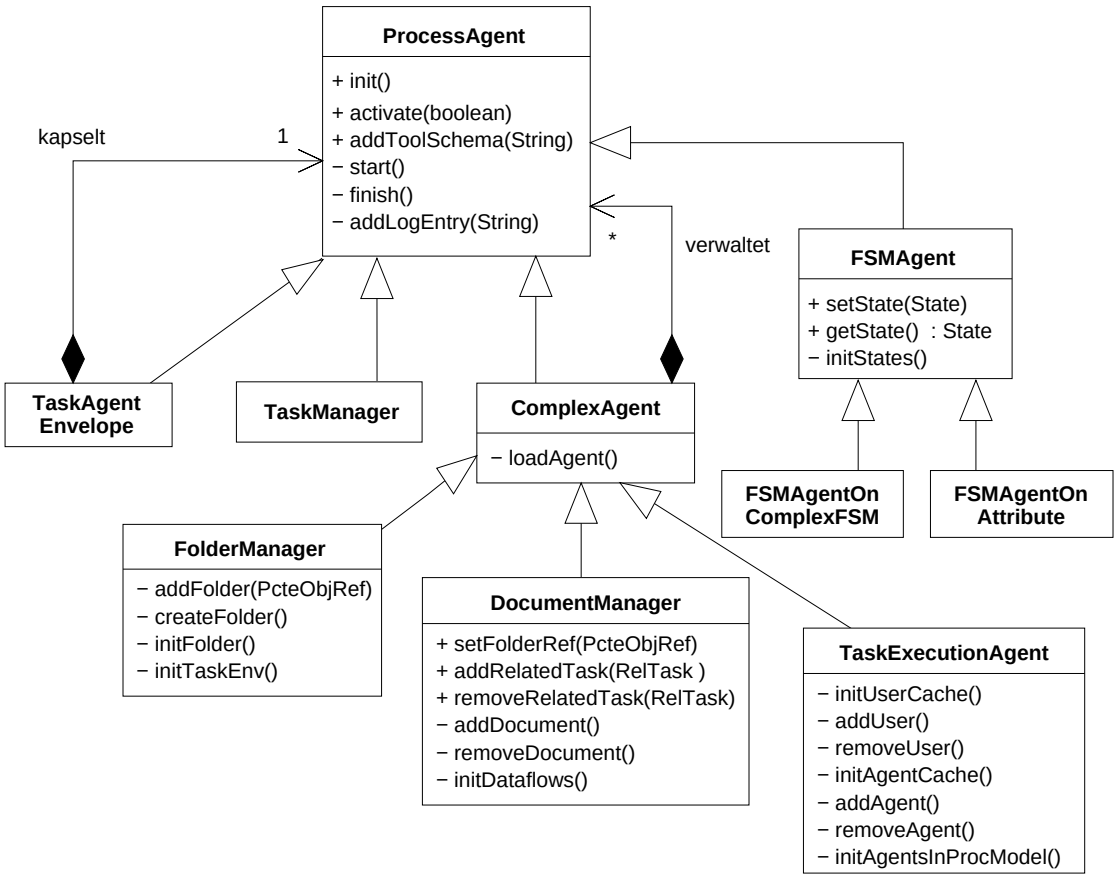


Abbildung 5.2: Ausschnitt aus der Klassenhierarchie für Prozeßagenten

angelegt werden. Außerdem lädt der TaskManager einen FSMAgent, mit dem er den Aufgabenzustand verwaltet.

FolderManager: Der FolderManager verwaltet die Dokumentordner von A. Nach dem Erzeugen der Aufgabe müssen die Ordner erzeugt, initialisiert und mit den nötigen Zugriffsrechten versehen werden. Welche Dokumentordner A besitzt, ist durch Werkzeugparameter spezifiziert, die dem Typ von A im Werkzeugschema der PE zugeordnet sind. Die Parameter enthalten auch den Zugriffsmodus für den Ordner, den Namen des Zustandsdiagramms, das die Dokumentzustände festlegt, und die Liste der Werkzeugschemata, mit denen die Werkzeuge zur Bearbeitung der enthaltenen Dokumente spezifiziert werden. Für jeden Dokumentordner von A lädt der FolderManager einen DocumentManager.

DocumentManager: Der DocumentManager verwaltet die Dokumente eines Dokumentordners D. An allen Dokumenten in D müssen die Zugriffsrechte passend gesetzt werden. Die ACL der Dokumente hängt ab von dem Zugriffsmodus des Ordners, dem Benutzer, der die Aufgabe ausführt, und der Definition weiterer Zugriffsrechte, mit denen anderen Benutzern der Zugriff erlaubt werden kann. Diese Rechte werden mit Werkzeugparametern am Ordnertyp definiert.

Außerdem realisiert der DocumentManager die Datenflüsse zwischen Aufgaben: Er erzeugt

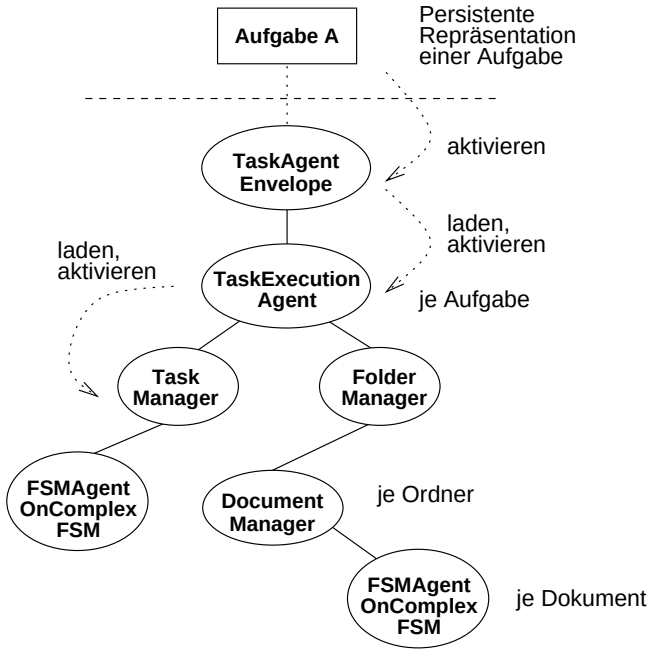


Abbildung 5.3: Beispiel für eine Agentenhierarchie in der Prozeßmaschine

Datenflußbeziehungen zu anderen Aufgaben und überwacht die Eigenschaften der Dokumente in D. Erfüllt ein Dokument die Bedingung an einem ausgehenden Datenfluß, wird es an die Zielaufgabe weitergeleitet, sofern der Ausführungsmodus eine automatische Weiterleitung festlegt (vgl. Abschnitt 4.2.4). Um die Datenflußbeziehungen verwalten zu können, muß der DocumentManager die 'Umgebung' von A kennen – der FolderManager überwacht Subaufgaben und Nachfolger von A und informiert die DocumentManager über Änderungen (addRelatedTask, removeRelatedTask).

FSMAgentOnComplexFSM: Ein FSMAgentOnComplexFSM implementiert Zustandsautomaten. Zustände und Transitionen werden hier durch ein Zustandsdiagramm definiert, das in Form von Objekten und Links in der Objektbank vorliegt. Der FSMAgentOnComplexFSM verwaltet eine transiente Kopie des Zustandsdiagramms und überwacht darin die Bedingungen an den Transitionen. Ist eine Bedingung erfüllt, wird der Zustand bei einem Agenten-Zustandsübergang sofort gewechselt; bei einem manuellen oder Werkzeug-Zustandsübergang wird die Transition lediglich als gültig markiert und kann vom Benutzer oder einem Werkzeug aktiviert werden (vgl. Abschnitt 4.2.5).

Die Bedingungen und Aktionen im Zustandsdiagramm werden ebenfalls von Komponenten der Prozeßmaschine implementiert: Im Zustandsdiagramm sind den Transitionen die Namen dieser Komponenten zugeordnet. Der FSMAgentOnComplexFSM lädt die Komponenten; die Bedingungen überwachen selbständig den Zustand der Objektbank und prüfen, ob sich Änderungen auf ihren Zustand auswirken. Sie nutzen hierzu den Benachrichtigungsmechanismus des OMS. Falls sich der Zustand ändert, wird die Änderung an den Zustandsautomaten propagiert, so daß dieser seinen Zustand wechseln kann. Bedingungen und Aktionen werden mit einfachen logischen Ausdrücken in Postfix-Notation verknüpft und parametrisiert:

Beispiel 5.1 (Zustandswechsel eines Dokuments) Voraussetzung für den Wechsel des Zustands von 'bearbeitet' nach 'konsistent' ist, daß das Dokument geprüft und fehlerfrei ist:

```
AttrHasValue(attr_name='checked', value=true)
AttrHasValue(attr_name='num_errors', value=0) &
```

Die Bedingungen vom Typ AttrHasValue bekommen als Parameter jeweils den Namen eines Attributs und den zu prüfenden Wert. Sie überwachen das Attribut und nehmen den Zustand *true* an, wenn das Attribut den angegebenen Wert hat. Ändert sich der Zustand der Und-Verknüpfung, wird die Änderung an den Zustandsautomaten propagiert, der wiederum die zugehörige Transition auslöst. Zu Implementierung werden die von Seite 121 bekannten Bedingungen verwendet.

□

Anpassung und Erweiterung

Wie bei den Werkzeugen wird auch die Zusammensetzung der Prozeßmaschine aus Komponenten durch Werkzeugschemata gesteuert. Das Werkzeugschema für die Prozeßmaschine enthält das Datenmodell zur Verwaltung der Prozeßinstanz und Werkzeugparameter, mit denen die Komponenten der PE spezifiziert und parametrisiert werden.

Mit prozeßspezifischen Komponenten können die Anforderungen des jeweiligen Prozesses berücksichtigt werden. Prozeßspezifische Bedingungen und Aktionen können in den Zustandsautomaten von Aufgaben und Dokumenten verwendet werden. Auch prozeßspezifische Agenten werden von der Prozeßmaschine geladen und ausgeführt. Sie können die Agenten des *genform*-Frameworks ergänzen oder auch ersetzen. Prozeßspezifische Komponenten entstehen durch Erweiterung vorhandener Komponenten, so daß der Implementierungsaufwand begrenzt wird.

5.2 Werkzeuge

Zur Steuerung und Überwachung des Entwicklungsprozesses werden *Prozeßwerkzeuge* benötigt: Neben der Prozeßmaschine müssen Benutzungsschnittstellen zur Verfügung stehen, mit denen sich die Benutzer über den Zustand des Prozesses informieren und den weiteren Fortschritt beeinflussen können:

- Manager erzeugen Aufgaben und bearbeiten die Beziehungen zwischen Aufgaben. Bei der Bearbeitung des Aufgabennetzes müssen die Konsistenzbedingungen aus dem Prozeßmodell beachtet werden – durch Einschränkung der möglichen Editieroperationen oder durch Prüfkommandos, die Fehler und Inkonsistenzen ermitteln.
- Manager weisen den Aufgaben Bearbeiter zu. Die Bearbeiter müssen die in der Aufgabendefinition geforderte Rolle besitzen.
- Manager ordnen den Aufgaben Dokumente zu und leiten Dokumente manuell über Datenflußbeziehungen an Folgeaufgaben weiter. Die Zuordnung von Dokumenten wird durch das Datenmodell der Dokumentordner eingeschränkt; zur Weiterleitung eines Dokuments muß es die Bedingung am Datenfluß erfüllen.

– Manager und Entwickler ändern den Zustand von Aufgaben auf Grundlage der zugeordneten Zustandsdiagramme.
– Manager und Entwickler ändern den Zustand von Dokumenten. Hier müssen die Zustandsdiagramme der Dokumentordner beachtet werden.

Durch die zahlreichen Prozeßwerkzeuge erhöht sich der Aufwand für den Bau der Meta-Umgebung sowie der konkreten PSEU. Die Nutzung von *genform* für den Bau der Prozeßwerkzeuge verringert die Kosten und erleichtert die Anpassung der Werkzeuge an den Prozeß sowie ihre Integration in die PSEU.

5.2.1 Generieren von Werkzeugen aus dem Prozeßmodell

Beim Generieren der Werkzeugschemata für die Prozeßwerkzeuge wird wie folgt vorgegangen:

– Aufgabentypen werden auf Objekttypen im Datenmodell des Werkzeugs abgebildet.
– Beziehungstypen zwischen Aufgaben werden auf Linktypen abgebildet.
– Für die verschiedenen Arten von Dokumentordnern werden Objekttypen erzeugt. Die Instanzen der Ordnertypen besitzen ausgehende Links, die auf die Dokumente verweisen.
– Die Eigenschaften der Prozeßelemente werden in Objekt- und Link-Attributen gespeichert.
– Weitere Eigenschaften zur Steuerung der Werkzeuge und der Prozeßmaschine werden in Werkzeugparametern kodiert.

Abbildung 5.4 zeigt einen Ausschnitt aus einem Prozeßmodell und das zugehörige Daten-

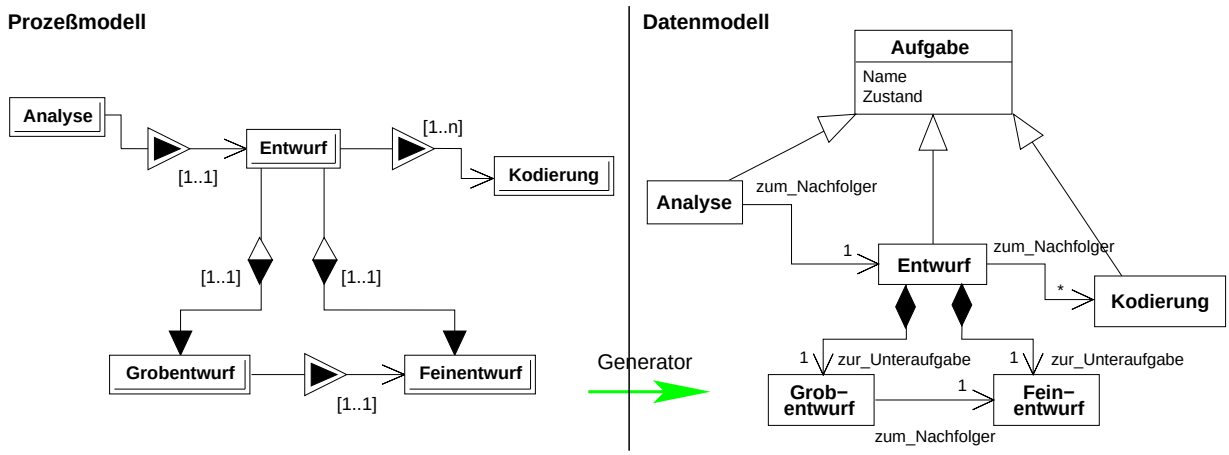


Abbildung 5.4: Generieren des Datenmodells für die Prozeßwerkzeuge

modell. Die Eigenschaften der verschiedenen Prozeßwerkzeuge werden durch folgende Werkzeugschemata spezifiziert:

1. Ein Basis-Werkzeugschema mit dem Datenmodell der Prozeßinstanz.
2. Ein Werkzeugschema für die Prozeßmaschine, das die Eigenschaften von Datenflußbeziehungen und die Namen der auszuführenden Prozeßagenten sowie ihre Parameter enthält.

3. Ein Werkzeugschema für das Planungswerkzeug, in dem das Datenmodell um Attribute zur Speicherung von Aufwänden und Terminen erweitert wird. Zusätzlich werden hier die Eigenschaften und Komponenten des Planungswerkzeugs mit Werkzeugparametern festgelegt.
4. Ein Werkzeugschema für ein Management-Werkzeug, das den Prozeßzustand darstellt und Kommandos zur Manipulation anbietet.
5. Einen Werkzeugschema für die Agenda der Entwickler.

5.2.2 Überwachung und Steuerung der Prozeßagenten

Die Agentenhierarchie existiert als transiente Struktur in der Prozeßmaschine. Ein Manager des Prozesses benötigt Informationen darüber, welche Agenten aktiv sind, welche Aktionen sie durchgeführt haben, und ob diese erfolgreich beendet wurden. Andererseits möchte er die Agenten auch beeinflussen, indem er ihnen Parameter übermittelt oder zusätzliche Agenten manuell in die Prozeßmaschine lädt.

Zur Überwachung und Manipulation der Agentenhierarchie müßte die Prozeßmaschine spezielle Schnittstellen anbieten, auf denen wiederum spezielle Werkzeuge arbeiten – dies bedeutet einen erhöhten Aufwand beim Bau der PSEU. Um den Aufwand zu verringern, wird in *genform* die Agentenhierarchie persistent in der Objektbank verwaltet. Ein Werkzeug (*PView*) zur Anzeige und Manipulation von Agenten wird mit einem Werkzeugschema beschrieben. Das Werkzeug kommuniziert mit der PE über das Repository: Es wird durch den Benachrichtigungsmechanismus über Änderungen informiert; umgekehrt überwacht die PE Änderungen, die der Benutzer mit dem Werkzeug durchführt.

Das Werkzeug zeigt die Aufgabenhierarchie mit den zugehörigen Agenten an (Abbildung 5.5; rechts ist die jeweils letzte Log-Ausgabe der Agenten zu sehen). Parameter, mit denen das Verhalten eines Agenten zur Laufzeit beeinflußt wird, können hier editiert werden: Die Parameter werden als Attribute der Objekte gespeichert, die die persistente Repräsentation des Agenten darstellen. Sie werden mit dem Werkzeug editiert und können vom Agenten gelesen werden. Beispiele für Parameter sind die Adressen von Nachrichtenempfängern eines Benachrichtigungsagenten oder die Art der Messungen, die ein Qualitätsagent durchführen soll.

Neben der Zuordnung von Prozeßagenten zu Aufgaben im Werkzeugschema können Prozeßagenten mit *PView* auch zur Laufzeit einer Aufgabe zugeordnet werden: Links von der Aufgabe zur Beschreibung eines Agenten veranlassen die PE, den Agenten zu laden. Die Links können mit *PView* editiert werden; die PE überwacht die Änderungen und paßt die Agentenhierarchie an. Abbildung 5.6 zeigt das zugehörige SDS. Das Attribut activation am Linktyp bestimmt die Aktivierung des Agenten: Er kann immer aktiv sein, nie oder abhängig von der Aktivierung der Aufgabe.

5.2.3 Interaktion zwischen Benutzern, Werkzeugen und Prozeß

Die Prozeßintegration der Werkzeuge einer CASE-Umgebung funktioniert in zwei Richtungen: Die Benutzer beeinflussen über ihre Werkzeuge den Zustand des Prozesses und den

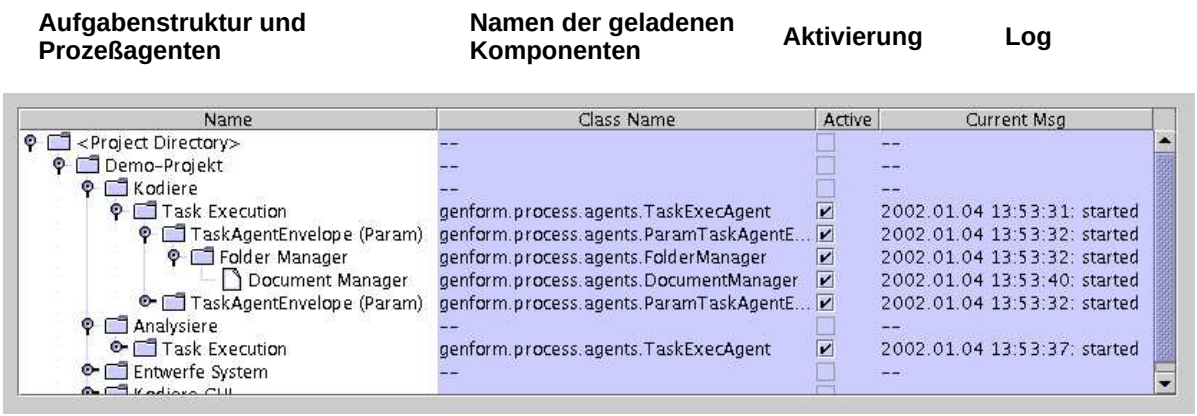


Abbildung 5.5: Anzeige von Aufgaben und geladenen Prozeßagenten in *PView*

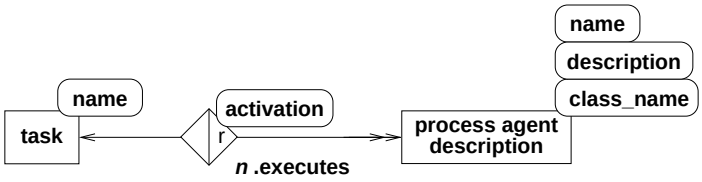


Abbildung 5.6: SDS für die Zuordnung von Agentenbeschreibungen zu Aufgaben

weiteren Fortschritt explizit – oder sie führen mit den Werkzeugen Aktionen aus, die sich implizit auf den Prozeßzustand auswirken. Umgekehrt müssen die Menge der verfügbaren Werkzeuge und der verfügbaren Dokumente sowie die Eigenschaften der Werkzeuge an die aktuelle Situation des Benutzers im Prozeß angepaßt werden.

Beeinflussung des Prozeßzustands

Die Liste der Aufgaben eines Entwicklers wird in seinem *Agenda-Werkzeug* angezeigt. Hier kann er die Bearbeitung einer Aufgabe beginnen, unterbrechen, wiederaufnehmen und beenden – also den Zustand der Aufgabe ändern und somit den Prozeßzustand direkt beeinflussen. Über die Aufgabe kann der Entwickler direkt auf zugeordnete Werkzeuge und Dokumente in seinem Arbeitskontext zugreifen.

Indirekt wird der Prozeßzustand durch die Änderung der Zustände von Dokumenten beeinflußt. Eine Reviewaufgabe ist erledigt (Zustand 'done'), sobald alle zugeordneten Diagramme geprüft wurden. Der Zustand der Aufgabe wird also vom Zustand der Dokumente abgeleitet. Für die Zustandspropagation ist die Prozeßmaschine zuständig.

Manuelle Zustandsübergänge (vgl. Abschnitt 4.2.5) können direkt im Entwurfswerkzeug ausgelöst werden: Der aufgabenspezifische Dokumentzustand erscheint als Eigenschaft des Dokuments und kann in einen erlaubten Zielzustand verändert werden. Die Zielzustände werden dabei von der Prozeßmaschine vorgegeben; sie verwaltet den Zustandsautomat des Dokuments. Automatische Zustandsübergänge erscheinen nicht in der Benutzungsschnittstelle, sondern werden direkt ausgelöst, etwa von einem Prüfwerkzeug. Somit können interaktive und nicht-interaktive Werkzeuge den Prozeß beeinflussen.

Beeinflussung der Werkzeuge

Im Prozeßmodell werden den Aufgabentypen Werkzeugschemata zugeordnet. Sie legen fest, welche Eigenschaften die CASE-Werkzeuge besitzen, die in einer Aufgabe verwendet werden. Zur Laufzeit werden die Werkzeuge auf zwei Arten beeinflußt: Über Typrechte an Attributen, Objekt- und Linktypen werden die ausführbaren Operationen eingeschränkt. Die Typrechte sind im Datenmodell des Werkzeugschemas definiert, das Verhalten des Werkzeugs wird damit durch seine aktuelle Sicht beeinflußt. Zum anderen werden die Eigenschaften der Ressourcen in der Objektbank ausgewertet. Dabei kommen weitere Mechanismen des OMS zum Einsatz:

– Zugriffsrechte gewährleisten, daß ein Benutzer nur die gemäß Prozeßmodell vorgesehenen Operationen auf den Daten im Repository ausführen kann. An jedem Dokument wird in der ACL vermerkt, welche Operationen welche Benutzer und Benutzergruppen auf dem Objekt ausführen können. Die Zugriffsrechte an Dokumenten werden im Prozeßmodell spezifiziert und von der Prozeßmaschine gesetzt.
– Durch Sperren werden parallele Werkzeugausführungen synchronisiert. Somit wird sichergestellt, daß keine Konflikte entstehen, wenn zwei Benutzer gleiche Daten bearbeiten. Dies könnte passieren, wenn mehrere Benutzer einer Aufgabe zugeordnet sind und gemeinsam ein Entwurfsdiagramm bearbeiten. Das Repository bietet einen feingranularen Sperrmechanismus [270] an, mit dem maximale Parallelität gewährleistet wird.

Die Werkzeugkomponenten in *genform* prüfen die Rechte- und Sperrsituation an Objekten, blenden nicht lesbare Einträge aus und deaktivieren nicht ausführbare Kommandos: Bestehen keine Schreibrechte an einem Dokument, werden die Kommandos zum Erzeugen und Löschen von Einträgen und die Eingabefelder im Formular deaktiviert – dadurch werden Werkzeuge implizit an den Prozeßzustand angepaßt.

Ein weiterer Vorteil ist, daß in der Implementierung der Werkzeuge weder das aktuelle Prozeßmodell noch das Meta-Modell oder die Mechanismen zur Beeinflussung des Prozeßzustands berücksichtigt werden müssen: Werkzeuge kommunizieren lediglich über gemeinsame Daten mit der PSEU, die Datenmodelle sind Parameter der generischen Werkzeuge.

5.2.4 Beispiel: Zustandsautomat für Dokumente

Die möglichen Zustände von Dokumenten werden mit Zustandsdiagrammen definiert. Die Zustandsdiagramme sind den Dokumentordnern zugeordnet, damit ordner- und aufgabenspezifisch. Abbildung 5.7 zeigt einen Ausschnitt aus dem Zustandsdiagramm für ein Analysedokument. Am Zustandsübergang sind die Ausdrücke für die Bedingung und die auszuführende Aktion notiert. Mehr zu den Meta-Werkzeugen, mit denen das Prozeßmodell und die Zustandsdiagramme definiert werden, in Abschnitt 5.4.

Zur Prozeßlaufzeit werden die Dokumentzustände von der Prozeßmaschine überwacht und manipuliert. In den Prozeß- und CASE-Werkzeugen werden die Zustände angezeigt und vom Benutzer geändert.

Repository. Beim Einrichten des Repositories für den Prozeß (Abschnitt 4.3.1) wird zunächst eine Datenstruktur angelegt, die die Struktur des Zustandsdiagramms und die Infor-

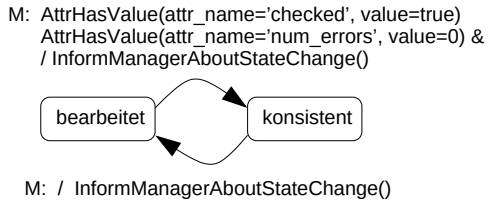


Abbildung 5.7: Ausschnitt aus dem Zustandsdiagramm eines Analysedokuments

mationen über Zustände und Transitionen enthält: Aus dem graphischen Dokument werden also Daten erzeugt, auf denen die Prozeßmaschine arbeitet.

In Abbildung 5.8 ist das zugehörige SDS graphisch dargestellt. Ein Zustandsdiagramm wird durch ein fsm-Objekt repräsentiert. Ein fsm enthält je ein State-Objekt für jeden Zustand. Transitionen werden durch Links des Typs transition modelliert. Am Link sind die Art der Transition und die Ausdrücke für Bedingungen und Aktionen gespeichert.

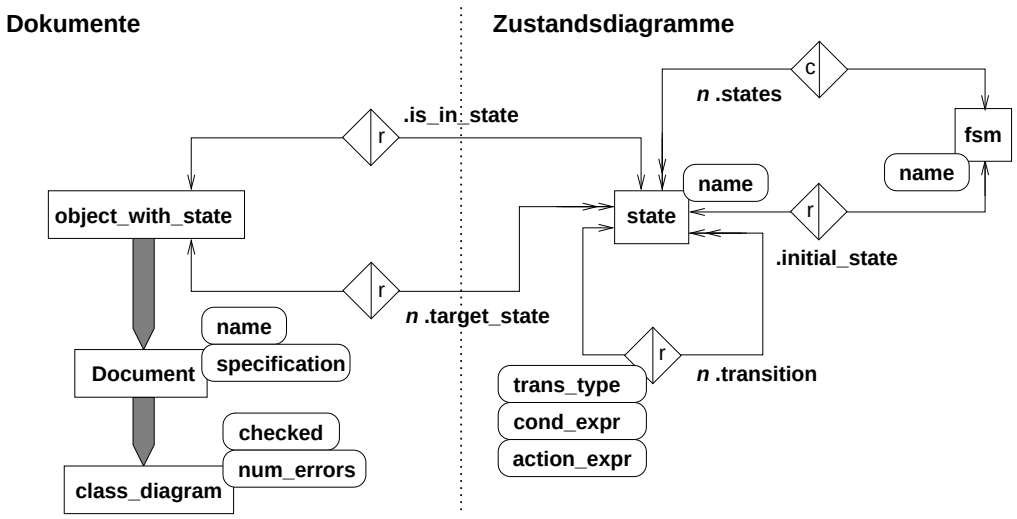


Abbildung 5.8: SDS für Dokumentzustände

Im linken Teil der Abbildung ist ein Ausschnitt aus dem Datenmodell für Dokumente zu sehen: Der aktuelle Zustand eines Dokuments wird durch einen Link des Typs is_in_state repräsentiert. Auf die möglichen Zielzustände verweisen Links des Typs target_state. In Abbildung 5.9 ist die Situation in der Objektbank für das Zustandsdiagramm aus Abbildung 5.7 skizziert.

Prozeßmaschine. Die Prozeßmaschine reagiert auf Änderungen im Prozeß. Da der Prozeßzustand im Repository verwaltet wird, kann die PE den Benachrichtigungsmechanismus des OMS zur Überwachung nutzen. Für die Verwaltung von Zuständen sind Prozeßagenten des Typs FSMAgent zuständig. Ein FSMAgent überwacht den Zustand genau eines Dokuments (vgl. Abbildung 5.3). In Abbildung 5.10 sind die Komponenten der PE dargestellt: Da das Zustandsdiagramm als komplexes Objekt in der Objektbank vorliegt, wird ein Agent vom Typ FSMAgentOnComplexFSM verwendet (Abschnitt 5.1.1).

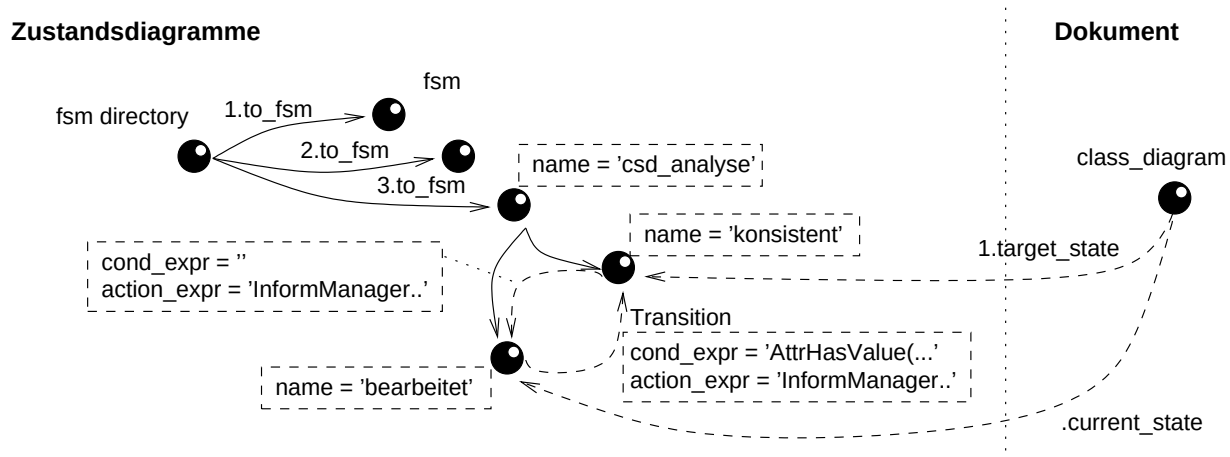


Abbildung 5.9: Situation in der Objektbank

Der Agent baut eine transiente Kopie des Zustandsdiagramms aus State- und Transition-Objekten auf. Für letztere werden Condition- und Action-Komponenten geladen: Sie implementieren die Überwachung und Auswertung des Prozeßzustands bzw. die Aktionen, die beim Zustandswechsel ausgeführt werden. Die Condition-Objekte überwachen dazu die Änderungen in der Objektbank. Im Beispiel müssen die Attribute von Klassendiagrammen überwacht werden – folglich muß das Datenmodell der Dokumente in der Sicht des FSMAgent enthalten sein. Er wird also als separater H-PCTE-Prozeß ausgeführt.

Ändert sich der Zustand einer Condition, wird die Änderung an eventuelle umgebende Bedingungen (im Beispiel die Und-Bedingung) propagiert und schließlich an die Transition. Das weitere Vorgehen hängt von der Art des Zustandsübergangs ab:

1. Bei einem Agenten-Zustandsübergang (Abschnitt 4.2.5) wird der Zustand sofort gewechselt, also der is_in_state-Link auf den neuen Zustand gesetzt und die target_state-Links angepaßt. Beim Zustandswechsel wird auch die zugeordnete Aktion ausgeführt.
2. Der manuelle Zustandsübergang im Beispiel wird vom Benutzer ausgelöst; die PE markiert lediglich den Zustandsübergang als gültig. Gültige Zustandsübergänge werden durch Links des Typs target_state auf die erlaubten Zielzustände markiert. Die zugeordnete Aktion wird erst ausgeführt, wenn der Benutzer den Zustand tatsächlich ändert[1].

Benutzungsschnittstelle. Der Benutzer benötigt eine Benutzungsschnittstelle, die den aktuellen Zustand des Dokuments anzeigt und die Änderung des Zustands erlaubt. Diese Benutzungsschnittstelle sollte möglichst in die Arbeitsumgebung des Benutzers integriert sein. Bei Entwicklern besteht die Arbeitsumgebung aus CASE-Werkzeugen – diese müssen angepaßt werden an

– den konkreten Prozeß: Welche Zustände hat ein Dokument, welche Zustandswechsel sind in der aktuellen Situation erlaubt, welche Kommandos können auf dem Dokument im aktuellen Zustand ausgeführt werden?

[1] Hierbei wird vorausgesetzt, daß jeweils nur maximal eine Transition von Zustand A nach Zustand B gültig sein kann, da sonst beim Zustandswechsel nicht festgestellt werden kann, welche Transition tatsächlich feuert.

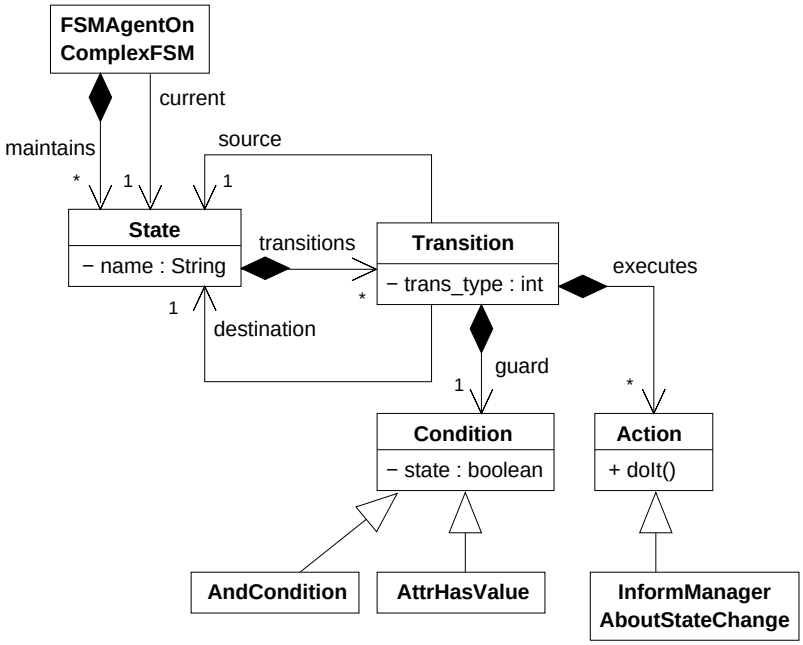


Abbildung 5.10: Komponenten der Prozeßmaschine

– die Art der Verwaltung von Dokumentzuständen: Wie werden Zustände repräsentiert, welche Änderungsoperationen sind nötig, welche speziellen GUI-Elemente werden gebraucht?
– die Prozeßmaschine: Wie werden Zustandsänderungen der PE mitgeteilt und wie informiert umgekehrt die PE das Werkzeug über eine Zustandsänderung oder die Änderung der möglichen Zielzustände?

In *genform* wird die Anpassung der Werkzeuge erleichtert: Es muß nur das Datenmodell erweitert werden, um Dokumentzustände anzeigen und ändern zu können. Darstellungen und verfügbare Kommandos werden mit Sichten und Zugriffsrechten von der Prozeßmaschine gesteuert. Abbildung 5.11 zeigt das GUI eines CASE-Werkzeugs, mit dem der Dokumentzustand geändert werden kann: Die *combo box* enthält den aktuellen Dokumentzustand und die möglichen Zielzustände. Ersterer wird durch den is_in_state-Link markiert; die Zielzustände sind die Zielobjekte der ausgehenden target_state-Links.

Für die Anzeige und Änderung des Zustands können also die generischen *genform*-Komponenten verwendet werden: Aus Sicht des Werkzeugs handelt es sich beim Dokumentzustand um eine ausgehende Referenz-Beziehung der Kardinalität eins (is_in_state). Die möglichen Zielobjekte werden aus einem Verzeichnis ausgewählt: Das Verzeichnisobjekt ist das Dokument selbst, die Einträge werden über Links des Typs target_state erreicht.

Das Werkzeug und die Prozeßmaschine kommunizieren über das Repository, da beide das Dokumentobjekt und die ausgehenden Links überwachen. Die *combo box* muß auf zwei Ereignisse reagieren: Bei einer Änderung des is_in_state-Links wird der aktuelle Eintrag angepaßt; ändern sich die target_state-Links, werden die Listeneinträge aktualisiert. Dabei müssen beachtet werden:

– Die Zugriffsrechte am Dokumentobjekt: Sie müssen das Erzeugen und Löschen ausgehender Links erlauben.

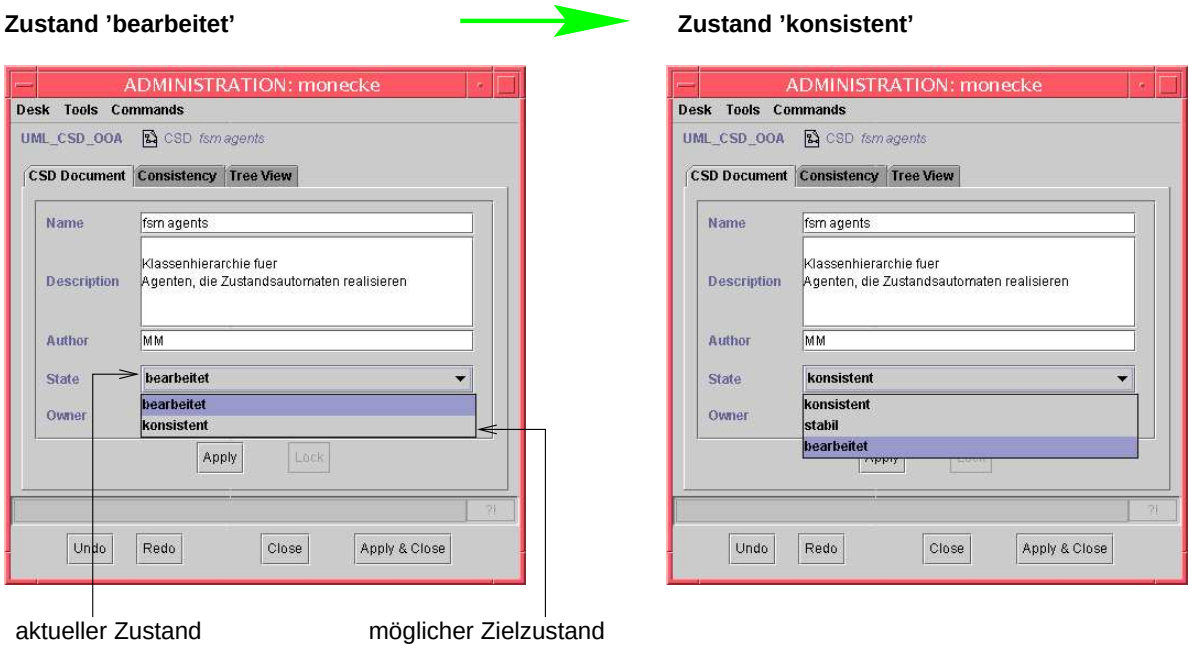


Abbildung 5.11: Benutzungsschnittstelle im CASE-Werkzeug zur Bearbeitung des Dokumentzustands

– Die Sperren am Dokumentobjekt und am is_in_state-Link.
– Die Typrechte an den Linktypen is_in_state und target_state.

Flexibilität des Ansatzes

Im beschriebenen Szenario werden Zustandsdiagramme zur Beschreibung der Dokumentzustände verwendet. Die Frage ist, welcher Aufwand entstehen würde, wenn Zustände auf andere Weise (etwa mit Petri-Netzen oder *State Charts* [154]) beschrieben würden.

– *Repräsentation des Zustandes:* Der aktuelle Zustand eines Dokuments wird im Repository durch einen Link auf ein Objekt markiert, das den Zustand darstellt. Alternativ kann der Zustand auch durch den Wert eines Aufzählungsattributs repräsentiert werden oder durch den Wert eines *string*-Attributs. Die erste Variante hat den Nachteil, das zur Änderung der möglichen Zustände das Datenbankschema geändert werden muß; die zweite hat den Nachteil, daß fehlerhafte Zustände (also ungültige Werte des Zustandsattributs) im Datenbankschema nicht verhindert werden können. Um die Repräsentation des Zustands zu ändern, muß also lediglich das Datenbankschema angepaßt werden.
– *Bestimmung des aktuellen Zustands:* Bei Zustandsdiagrammen werden Zustandswechsel ausgeführt, sobald die Bedingung an einer vom aktuellen Zustand ausgehenden Transition wahr wird. Die Logik des Zustandsautomaten ist im FSMAgent implementiert, die konkreten Bedingungen und Aktionen in weiteren Komponenten. Der FSMAgent kann durch einen Agenten ersetzt werden, der eine andere Zustandsberechnung implementiert. Bei einer Änderung des Datenbankschemas, mit dem Zustände verwaltet werden, muß auch der Agent angepaßt werden. Ein Austausch des Agenten ist durch eine Änderung der Konfiguration der Prozeßmaschine leicht möglich.

– *Benutzungsschnittstelle:* Die Benutzungsschnittstelle zur Anzeige und Änderung von Zuständen muß an das Datenbankschema angepaßt sein. Wird der Zustand in einem Aufzählungsattribut verwaltet, muß dieses bearbeitet werden können. *genform* enthält ein GUI-Element, mit dem ein Aufzählungsattribut in einer *combo box* editiert werden kann. Problematisch ist bei dieser Lösung, daß jeweils nur eine Teilmenge der möglichen Werte des Aufzählungsattributs tatsächlich wählbar ist – nämlich nur die erlaubten Zielzustände. Die erlaubten Werte werden daher in einem weiteren Attribut gespeichert und vom GUI-Element ausgewertet.

Die hohe Flexibilität wird also erreicht, da die Prozeßmaschine und die GUI der Werkzeuge flexibel konfigurierbar sind. Kritisch sind Änderungen des Datenbankschemas, da sie sich sowohl auf die GUI als auch auf die Zustandsberechnung auswirken.

5.3 Andere Ansätze

PEACE+ [7] ist ebenfalls agentenbasiert; der Prozeßzustand wird auch persistent in einem OMS verwaltet. Eine *Reflector*-Komponente sorgt dafür, daß der Zustand im OMS und die Zustände der Agenten konsistent bleiben. In *ADEPT* [182] steht die Autonomie der Agenten im Vordergrund: Der Prozeß wird durch die Agenten und ihre Beziehungen festgelegt; es gibt keine Beschreibung der Aufgaben. Die Agenten kommunizieren mit einer speziellen Sprache, dadurch wird der Ansatz leichter erweiterbar – auch um heterogene Agenten. Die Kommunikationsinfrastruktur ist *CORBA* [28]. *CORBA* wird auch von Emmerich [98] für den Bau verteilter und heterogener SEU vorgeschlagen, die auch mehrere Repositories enthalten können. Problematisch ist hier der hohe Implementierungsaufwand, auch für die Änderungspropagation, die von Hand in den CORBA-Klassen implementiert werden muß.

Joeris [190] verwendet einen *task graph* mit Aufgaben, Kontroll- und Datenflüssen. Die Aufgaben werden nicht von einer zentralen Prozeßmaschine koordiniert. Statt dessen ist, wie in *genform*, jeder Aufgabe ein Agent zugeordnet, der den Prozeßfortschritt überwacht, Entscheidungen trifft und selbständig reagiert. Agenten sind allerdings nicht als austauschbare und erweiterbare Komponenten realisiert: Ihr Verhalten wird mit ECA-Regeln spezifziert. Somit ist die Flexibilität bei der Erweiterung und Adaptierung eingeschränkt.

Die PSEU aus dem *SPADE*-Ansatz [19] nutzt *DEC FUSE* für die Werkzeugintegration. Eine SPADE-Komponente empfängt eingehende Nachrichten und modifiziert den Prozeßzustand. Umgekehrt kann in speziellen Transitionen (*black transitions*) der Nachrichtenversand programmiert werden. Voraussetzung für die Integration eines Werkzeugs ist natürlich, daß es kompatibel zu *DEC FUSE* ist. Nachteilig ist auch, daß die Kommunikation aufwendig und fehleranfällig programmiert werden muß.

AHEAD [179] hat wie *genform* das Ziel, den Bau von Management-Werkzeugen zu vereinfachen. In *AHEAD* werden Prozesse mit UML-Diagrammen beschrieben [290] und daraus *PROGRES*-Regeln erzeugt, woraus wiederum C-Quelltext für die Werkzeuge generiert wird. Somit wird der Aufwand verringert, da nur "kleine Teile" von Hand implementiert werden müssen, und die Korrektheit der Werkzeuge bezogen auf das Prozeßmodell sichergestellt. Die letzten Aussagen gelten auch für *genform*. Allerdings werden in *AHEAD* kommerzielle Werkzeuge, kommerzielle und freie Bibliotheken sowie zahlreiche Sprachen beim Bau der

Management-Umgebung verwendet. *genform* ist also der schlankere Ansatz und einfacher zu warten und zu erweitern.

P-Root [48] ist eine objektorientierte Erweiterung von PCTE und dient als Basis für PSEU. In *P-Root* werden Operationen im Datenbankschema definiert; in unterschiedlichen Sichten können also Typdefinitionen gemeinsam genutzt werden, während das Verhalten vom aktuellen Arbeitsschema abhängt. Außerdem werden Arbeitsbereiche unterstützt. Die Datenmodelle sind hier allerdings grobgranular: Ein Dokument entspricht einem Objekt, das zur Bearbeitung in den Arbeitsbereich eines Entwicklers kopiert wird. Somit können bei der Prozeßmodellierung die Eigenschaften von Dokumenten und ihre innere Struktur nicht berücksichtigt werden. In *genform* werden die Dokumente in der Produktstruktur verwaltet, der Prozeßzustand enthält lediglich Verweise auf Dokumente und Dokumentteile. Somit kann auch stets über die Produktstruktur oder über die Aufgaben auf die Dokumente zugegriffen werden.

5.4 Meta-Werkzeuge

Meta-Werkzeuge unterstützen den Werkzeugentwickler beim Bearbeiten der Werkzeug- und Prozeßspezifikationen. *genform* enthält eine Sammlung von Meta-Werkzeugen, die *genform tool builder tools* (*gtb*).

gtb umfaßt:

- Funktionen zur textuellen und graphischen Darstellung von Werkzeug- und Prozeßspezifikationen
- Editierfunktionen zur Bearbeitung der Spezifikationen
- Konsistenzprüfungen, mit denen die Korrektheit der Spezifikationen geprüft wird
- Transformationsfunktionen, mit denen lauffähige Werkzeuge aus den Spezifikationen erzeugt werden
- Möglichkeiten zur Wiederverwendung von Teilen der Werkzeug- und Prozeßspezifikationen

Die *gtb*-Werkzeuge unterscheiden sich in ihren Funktionen also nicht wesentlich von anderen CASE-Werkzeugen. Somit liegt es nahe, *genform* zum Bau der Werkzeuge zu nutzen – und so auch die Flexibilität des Ansatzes zu prüfen. Abbildung 5.12 zeigt einige Werkzeugfenster.

Die *gtb*-Werkzeuge werden mit Werkzeugschemata spezifiziert. Diese enthalten die Datenmodelle der Werkzeuge und die Werkzeugeigenschaften in Form von Werkzeugparametern. Die Werkzeuge bestehen aus generischen Komponenten des *genform*-Frameworks. Zusätzlich werden werkzeugspezifische Komponenten benötigt für

- die graphische Darstellung von Daten- und Prozeßmodellen.
- Kommandos zur Konsistenzprüfung und zum Export der Spezifikationen.

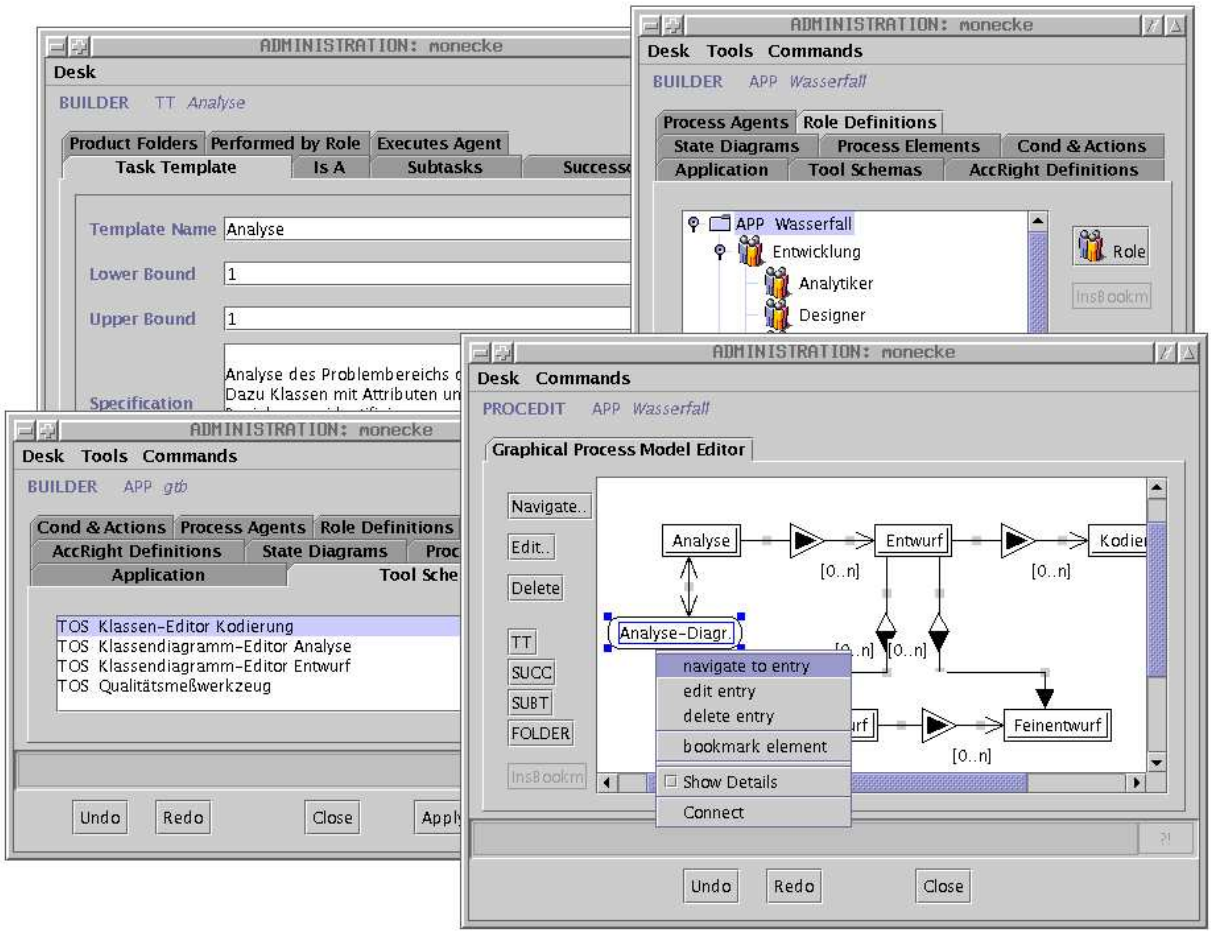


Abbildung 5.12: Einige *gtb*-Werkzeuge

5.4.1 Datenmodelle

Das Datenmodell von *gtb* entspricht dem *genform*-Meta-Modell: Die Dokumente, die mit *gtb* bearbeitet werden, sind Modelle für *genform*-Werkzeuge. Sie werden im Entwurfsrepository verwaltet, auf dem die *gtb*-Werkzeuge arbeiten (vgl. Abschnitt 3.1.6 auf Seite 103).

Mit *gtb* werden *Anwendungen* spezifiziert – integrierte Werkzeugsammlungen, die mit verschiedenen Sichten auf einem gemeinsamen Datenmodell arbeiten. Beispiele für solche Anwendungen sind PSEU. Abbildung 5.13 enthält einen Ausschnitt aus dem SDS, mit dem Anwendungen in *gtb* verwaltet werden.

Eine Anwendung (Objekttyp application) besteht aus

1. einer Menge von *Werkzeugschemata*, die die enthaltenen Werkzeuge beschreiben (tool_schema). Das Datenmodell der Anwendung ergibt sich aus der Vereinigung der Sichten, die durch die Werkzeugdatenmodelle definiert werden. Zwischen den einzelnen Datenmodellen müssen Beziehungen existieren, die die verschiedenen Werkzeuge verbinden. Über diese Beziehungen kann der Benutzer zu anderen Dokumenten navigieren und Werkzeuge zur Bearbeitung starten. Das SDS zur Verwaltung von Werkzeugschemata ist in Abbildung

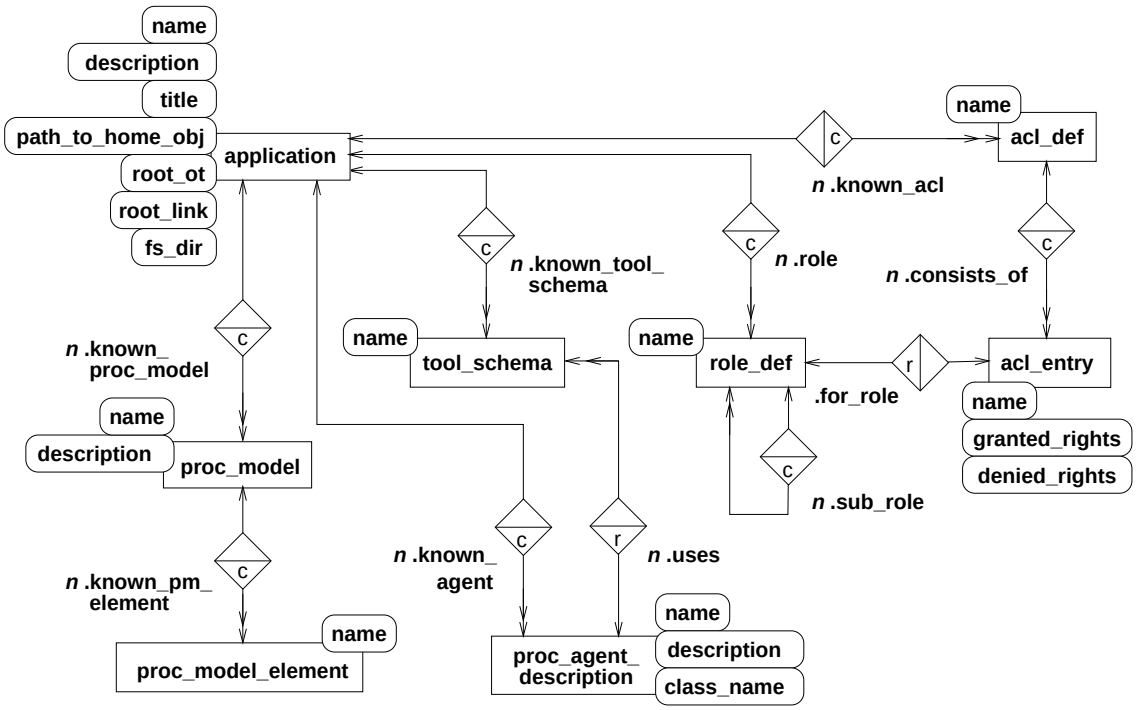


Abbildung 5.13: SDS für Anwendungen in *gtb*

5.14 dargestellt.

2. einer Menge von *Prozeßmodellen* (proc_model), die Aufgabentypen und ihre Beziehungen vorschreiben. Die Subtypen von proc_model_element sind in Abbildung 5.15 eingezeichnet. Eine generierte Anwendung enthält Prozeßwerkzeuge für die spezifizierten Prozeßmodelle.
3. einer Menge von *Rollendefinitionen* (role_def). Rollen können hierarchisch sein.
4. einer Menge von *Zugriffsrechte-Definitionen* (acl_def). Jede Zugriffsrechte-Definition entspricht einer ACL, ordnet also Benutzern oder Rollen erlaubte und verbotene Zugriffsmodi zu.
5. einer Menge von *Agentenbeschreibungen* (proc_agent_description). Eine Agentenbeschreibung enthält Name und Spezifikation eines Prozeßagenten, sowie den Namen der Komponente, die den Agenten implementiert (class_name). Somit wird auch die Zusammensetzung der Prozeßmaschine durch die Spezifikation festgelegt.

In den Attributen des Objekttyps application wird der Name des Verzeichnisses im Dateisystem gespeichert, in dem sich die anwendungsspezifischen Werkzeugklassen befinden (fs_dir). Unterhalb dieses Wurzelverzeichnisses wird eine Verzeichnishierarchie erzeugt, in der die Werkzeugkomponenten abgelegt werden. Anhand des Pfads zum Wurzelobjekt der Anwendung in der Objektbank (path_to_home_object) und dem Typ des Wurzelobjekts (root_ot) kann der generische Werkzeugkern auf die Daten der Anwendung zugreifen.

Werkzeugschemata

Mit dem SDS aus Abbildung 5.14 werden Werkzeugschemata verwaltet – es beschreibt die Struktur des Objekttyps tool_schema aus Abbildung 5.13.

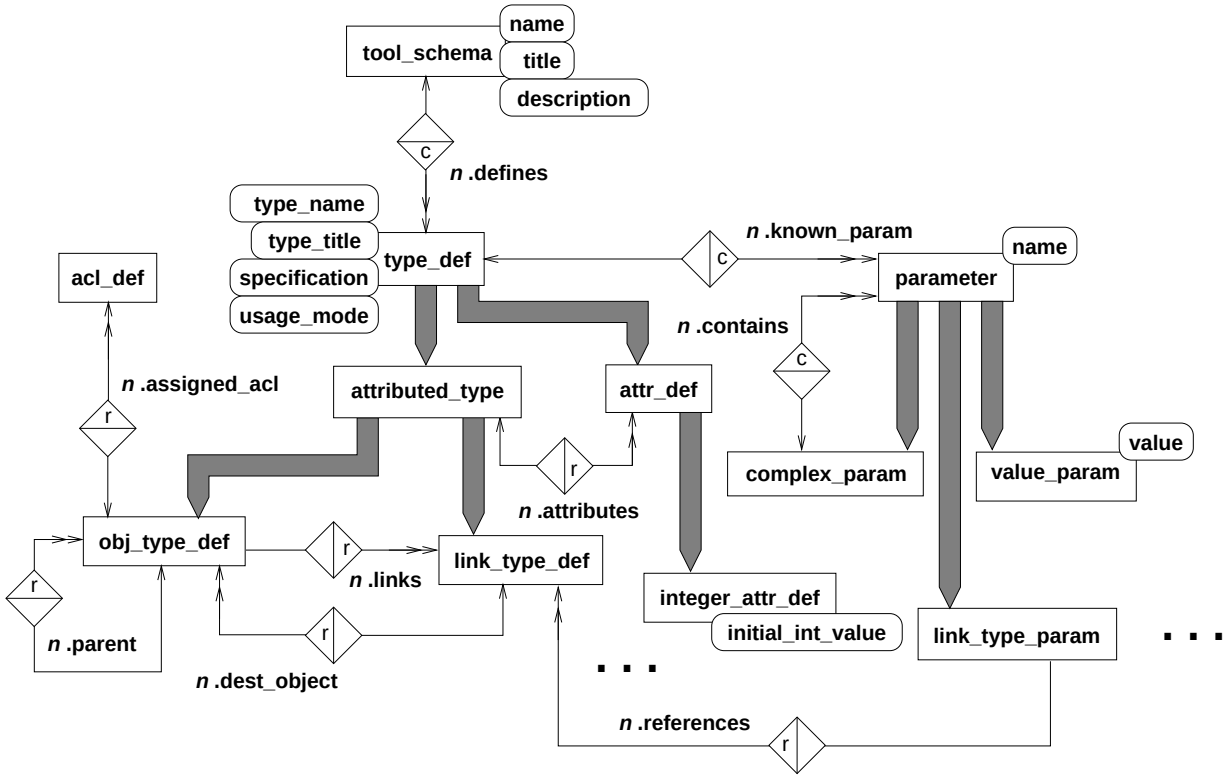


Abbildung 5.14: SDS für Werkzeugschemata

Da zur Beschreibung von Dokumenten eine Teilmenge der Modellierungskonzepte aus dem PCTE-Datenmodell verwendet wird, entspricht das Datenmodell hier dem Meta-SDS von H-PCTE. Bei den Attributen werden Typen für die verschiedenen Wertebereiche unterschieden (in der Abbildung beispielhaft integer_attr_def).

Die Beziehungen zwischen Typdefinitionen und Werkzeugparametern (known_param) sowie zwischen Objekttypen und den zugeordneten Zugriffsrechten (assigned_acl) stellen eine Erweiterung gegenüber dem PCTE-Datenmodell dar: Über erstere können Attributen, Objekt- und Linktypen Werkzeugparameter zugeordnet werden. Werkzeugparameter werden unterschieden in Wertparameter (value_param) und komplexe Parameter (complex_param). Spezielle Parameter dienen zur Spezifikation von Attributen, Objekt- oder Linktypen – sie besitzen Referenzen auf die Typdefinitionen im Datenmodell. In der Abbildung ist der Parametertyp link_type_param eingezeichnet. Parameter dieses Typs können beispielsweise verwendet werden, um die ausgehenden Links eines Objekts zu spezifizieren, die in einer Baumdarstellung angezeigt werden sollen.

Prozeßmodelle

Prozeßmodelle werden mit dem SDS aus Abbildung 5.15 verwaltet. Ein Prozeßmodell besteht aus Aufgaben und Beziehungen zwischen Aufgaben – Beziehungen zu Unteraufgaben (subtask_conn) und zu Nachfolgern (succ_conn). Letztere besitzen das Attribut succ_type, in dem die Art der Nachfolgerbeziehung (*sequential* oder *concurrent*) gespeichert wird. Über einen Link des Typs performed_by kann einer Aufgabe die Rolle zugeordnet werden, die der Bearbeiter besitzen muß.

Über Links des Typs executes werden die Beschreibungen von Agenten zugeordnet, die bei der Ausführung der Aufgabe aktiviert werden sollen. Am Linktyp executes speichert das Attribut activation, wann der Agent aktiviert werden soll – bei Aktivierung der Aufgabe, immer oder nie.

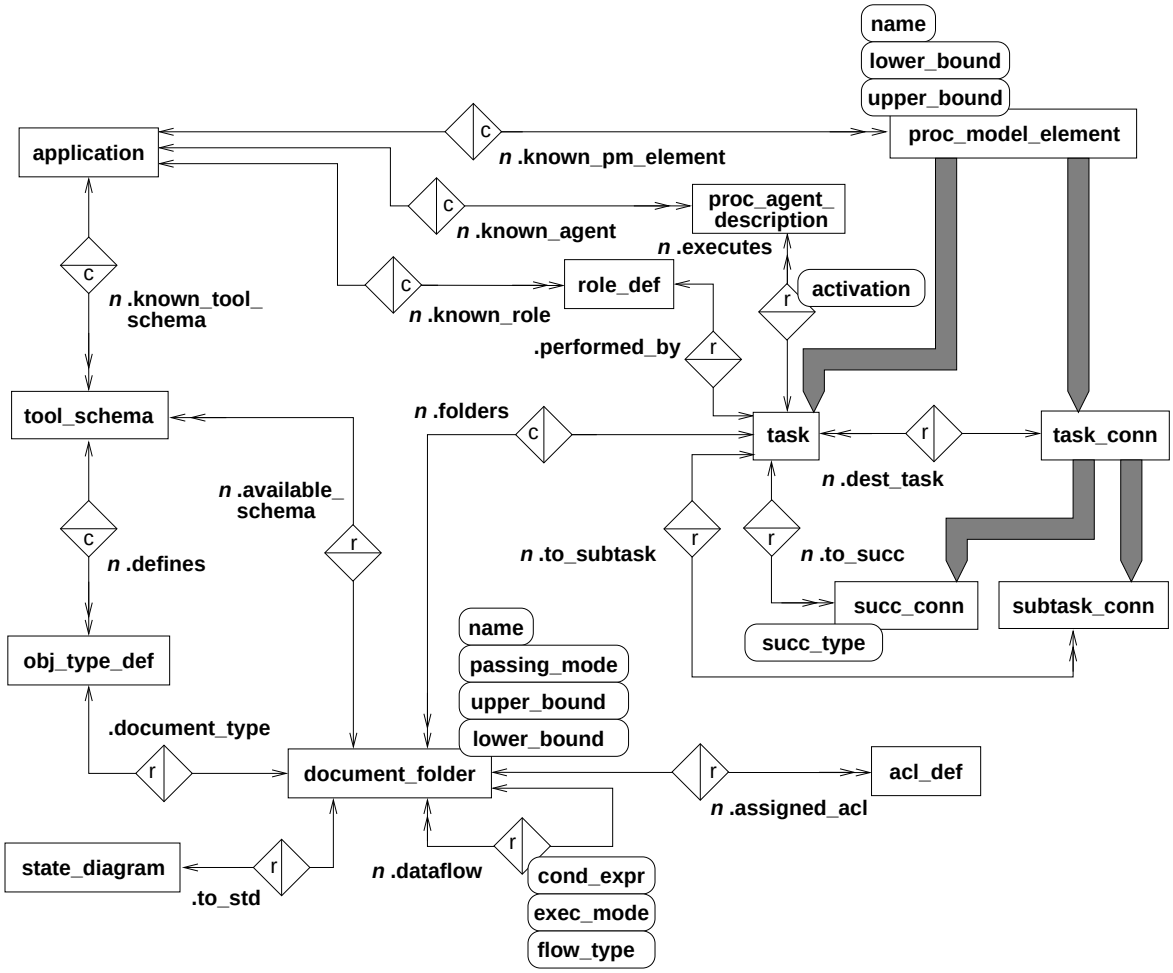


Abbildung 5.15: SDS für Prozeßmodelle

Dokumentordner werden durch den Objekttyp document_folder modelliert und den Aufgaben über Links des Typs folders zugeordnet. Ein Dokumentordner besitzt Attribute für seinen Namen, die Übergabeart, sowie eine untere und obere Grenze für die Zahl der enthaltenen Dokumente.

dataflow-Links markieren Datenflüsse zwischen Ordnern; sie besitzen einen Ausdruck für die Bedingung, die Dokumente erfüllen müssen (cond_expr), einen Ausführungsmodus (exec_mode manuell, Werkzeug, Agent) und einen Datenflußtyp (flow_type). Dieser legt fest, ob die Dokumente aus dem Ausgangsordner in den Zielordner übertragen oder zum Verzeichnis des Zielordners werden, oder ob das Verzeichnis des Ausgangsordners zum Dokument im Zielordner wird (*contents, decompose, recompose*). Das zugeordnete state_diagram definiert die Zustände der Dokumente im Ordner; der document_type-Link verweist auf den Typ der enthaltenen Dokumente im Dokument-Datenmodell.

Über Links des Typs assigned_acl können einem Dokumentordner ACL zugeordnet werden.

Diese ACL werden an allen enthaltenen Dokumenten gesetzt. Mit den ACL kann auch anderen Benutzern und Benutzergruppen der Zugriff auf die enthaltenen Dokumente erlaubt werden.

5.4.2 Initialisieren des Produktionsrepositories

Aus den Spezifikationen, die mit den *gtb*-Werkzeugen bearbeitet wurden, muß eine ausführbare PSEU erzeugt werden. Zum Bau der PSEU mit *genform* werden also benötigt:

1. *Werkzeugschemata* für die Werkzeuge der PSEU. Sie bestehen aus dem Datenmodell der Werkzeuge und Werkzeugparametern. Um die Daten im OMS verwalten zu können, muß das Datenmodell in Form von SDS in der Objektbank vorliegen.
2. *Generische* *genform*-Komponenten, aus denen die Werkzeuge zusammengesetzt werden. Die Komponenten sind im Framework enthalten und werden zur Laufzeit vom generischen Werkzeugkern geladen. Als Parameter für die generischen Komponenten dienen die Werkzeugschemata der PSEU.
3. *Werkzeugspezifische* Komponenten, die an die Anforderungen der konkreten PSEU angepaßt sind. Diese Komponenten werden vom Werkzeugentwickler implementiert und im ausführbaren Format bereitgestellt.

Die *gtb*-Werkzeuge bieten eine Unterstützung beim ersten Punkt: Der Bearbeitung und Prüfung von Spezifikationen und der Transformation der Spezifikationen in ein Format, das von den generischen Werkzeugkomponenten interpretiert werden kann. Außerdem kann mit *gtb* das Produktionsrepository initialisiert werden, auf dem die PSEU arbeitet (vgl. Abschnitt 3.1.6). Bei dieser Initialisierung werden

- Benutzergruppen angelegt. Sie repräsentieren die Rollen des Prozeßmodells.
- SDS in der Objektbank eingerichtet. Sie enthalten die Typdefinitionen, mit denen die Werkzeuge in den verschiedenen Sichten arbeiten.
- Zustandsdiagramme erzeugt, die die Zustände von Dokumenten und Aufgaben beschreiben.
- das Wurzelobjekt für die Anwendung erzeugt. Alle Daten der Anwendung sind über Links von diesem Objekt aus erreichbar.
- Verzeichnisse angelegt, die Daten aus dem Prozeßmodell enthalten, auf die die Werkzeuge zugreifen müssen, etwa die Liste der verfügbaren Prozeßagenten.

Die *gtb*-Werkzeuge erzeugen diese Daten im Produktionsrepository nicht direkt, sondern exportieren Transportdateien, die mit Initialisierungswerkzeugen in das Produktionsrepository eingespielt werden (Abbildung 5.16). Datenbankschemata werden als textuelle SDS-Definitionen exportiert und können mit einem Administrationswerkzeug direkt in die Objektbank übersetzt werden. Alternativ könnten auch die OMS-Operationen zur Schema-Manipulation vom Initialisierungswerkzeug direkt aufgerufen werden – allerdings wäre der Implementierungsaufwand für diesen Ansatz höher, da die Schema-Operationen von der OMS-Zugriffsschicht in *genform* bisher nicht unterstützt werden.

Für Benutzergruppen und Zustandsdiagramme werden Dateien im XML-Format exportiert, die Informationen mit Parsern extrahiert und in Aufrufe der OMS-Operationen umgesetzt. Somit müssen die *gtb*-Werkzeuge und das Entwurfsrepository beim Einrichten des Produktionsrepositories nicht verfügbar sein.

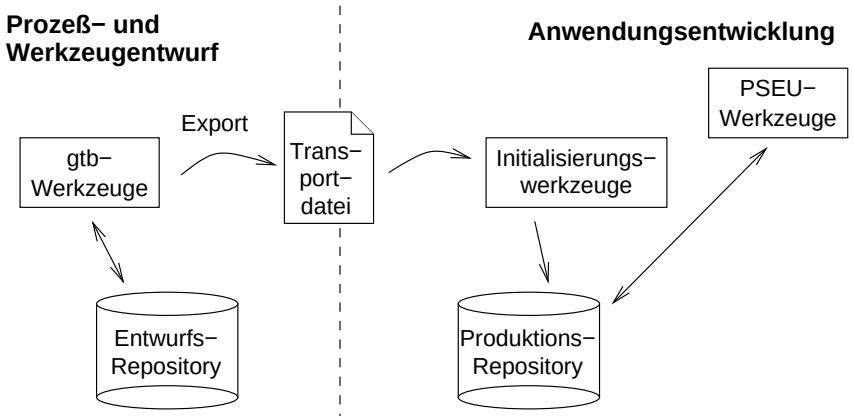


Abbildung 5.16: Initialisieren des Produktionsrepositories

Werkzeugschemata werden als Java-Quelltexte exportiert und in einer Verzeichnisstruktur unterhalb des Wurzelverzeichnisses der Anwendung abgelegt. Der Name des Wurzelverzeichnisses wird im Attribut fs_dir aus dem SDS in Abbildung 5.13 gespeichert. Der Vorteil dieses Verfahrens ist, daß kein Parser für die Werkzeugschemata implementiert werden mußte und die internen Datenstrukturen zur Werkzeuglaufzeit schneller aufgebaut werden können als bei Verwendung eines Parsers.

5.4.3 Prozeßunterstützung für den Werkzeugentwurf

Da die *gtb*-Werkzeuge mit *genform* gebaut sind, kann in *gtb* auch eine Prozeßunterstützung für den Prozeß- und Werkzeugentwurf realisiert werden. Hierzu müssen zunächst

- die Rollen beim Werkzeugentwurf definiert werden, etwa für die Datenmodellierung, die Spezifikation der Werkzeugeigenschaften, die Implementierung von Werkzeugkomponenten, die Dokumentation oder den Test.
- die Arbeitsschritte beim Werkzeugentwurf festgelegt und so der Entwurfsprozeß strukturiert werden.
- Anforderungen an die einzelnen Dokumente festgelegt und in Form von Zuständen definiert werden. Bedingungen an den Zustandsübergängen prüfen, ob die Anforderungen des Zielzustands auch tatsächlich erfüllt werden. Zur Prüfung der Dokumenteigenschaften müssen passende Tests implementiert werden, etwa um ein Dokument auf Fehlerfreiheit, Konsistenz und Vollständigkeit zu prüfen. Durch Aktionen an den Zustandsübergängen können Tätigkeiten automatisiert werden, etwa das Erzeugen der XML-, SDS- und Quelltextdateien oder die Benachrichtigung der Beteiligten über den Prozeßfortschritt.

Abbildung 5.17 zeigt ein einfaches Prozeßmodell für den Werkzeugentwurf. Das Modell ist zunächst nicht sehr restriktiv. Nach einer Erprobung des Prozeßmodells könnten die Vor-

schriften verschärft werden, etwa um häufige Fehler zu verhindern. Das Prozeßmodell hat folgende Vorteile:

- Alle Schritte beim Werkzeugentwurf sind notiert. Dadurch wird gewährleistet, daß kein wichtiger Schritt vergessen wird und auch Neulinge den Ablauf überblicken können. Durch die Zuordnung von Werkzeugen und Dokumenten zu den Aufgaben finden die Werkzeugentwickler stets die passende Arbeitsumgebung vor und können direkt auf die benötigten Daten zugreifen.
- Die Bearbeiter der einzelnen Aufgaben können leicht ermittelt werden, was die Kommunikation zwischen den Beteiligten erleichtert.
- Der Prozeß kann leichter überwacht werden, da die Zustände der einzelnen Dokumente und Arbeitsschritte direkt abgelesen werden können. Zusätzlich können die erforderlichen Eigenschaften von Zwischenprodukten definiert und im Prozeß kontrolliert werden, was den Kommunikationsaufwand im Prozeß und die Zahl erforderlicher Überarbeitungen verringert.

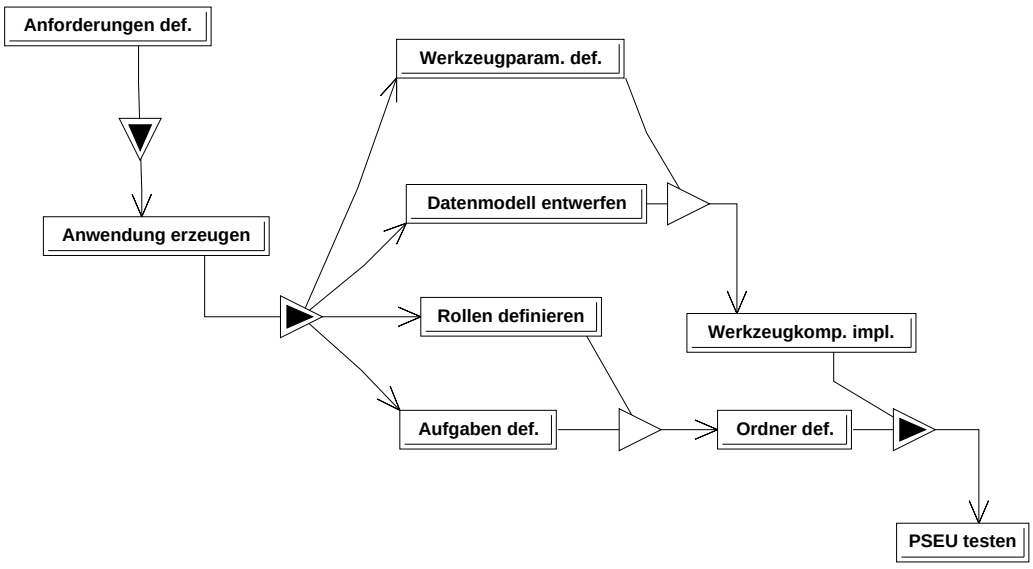


Abbildung 5.17: Prozeßmodell für den Werkzeugentwurf mit *gtb*

Primär wird der Prozeß jedoch durch die Beteiligten gesteuert. Daher wurde auch auf Rückkopplungsschleifen verzichtet. Die Abhängigkeiten zwischen Aufgaben und die Überwachung der Dokumenteigenschaften erleichtern aber die Kontrolle des Prozeßfortschritts und tragen so zu einer kostengünstigeren Produktion von Werkzeugen hoher Qualität bei.

Auf Dokumentordner wurde im Beispiel verzichtet. Sinnvoll wäre ein Ordner für die Anforderungsdefinition, die über einen Datenfluß an die Aufgabe Anwendung erzeugen weitergeleitet würde. Diese Aufgabe würde die erzeugte Anwendung in einem Ordner verwalten und an die Folgeaufgaben weiterleiten. Alle Folgeaufgaben könnten so direkt auf das Anwendungsobjekt zugreifen.

Bei einer feingranulareren Steuerung der Datenflüsse würde das Anwendungsobjekt als Verzeichnis der Ordner der Folgeaufgaben aufgefaßt. In den Aufgaben würden dann nur Komponenten der Anwendung (etwa Datentyp-Definitionen, Aufgaben) bearbeitet. Allerdings

werden so die Zugriffsmöglichkeiten des einzelnen Entwicklers stärker eingeschränkt. Die verfügbaren Darstellung, Kommandos und Änderungsoperationen werden durch die Werkzeugsichten festgelegt, die den Dokumentordnern zugeordnet sind.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Inhalt und Beitrag

Die Produktivität von Software-Entwicklern und die Qualität der entwickelten Produkte kann durch PSEU maßgeblich gefördert werden: Die enthaltenen CASE-Werkzeuge fördern ein formales Vorgehen und tragen durch Maßnahmen zur Konsistenzsicherung und zum Ermitteln von Mängeln in den erstellten Dokumenten zur Steigerung der Qualität und zur Verringerung von Fehlern bei. Die Modellierung des Software-Entwicklungsprozesses, die Planung, Steuerung und Überwachung der Prozeßdurchführung können dazu beitragen, daß aus Sicht der Qualitätssicherung wichtige Schritte auch wirklich ausgeführt, aus Sicht des Managements Abweichungen vom Plan schnell erkannt werden können, und der aktuelle Prozeßzustand präziser beurteilt werden kann.

Allerdings ist es schwierig, eine geeignete PSEU zu finden oder zusammenzustellen: CASE-Werkzeuge sind oft zu komplex, nicht an die jeweiligen Anforderungen angepaßt (und anpaßbar) und nicht mit einem Mechanismus zur Steuerung des Entwicklungsprozesses integriert – was die Effizienz beim Einsatz und die Motivation dazu reduziert und somit Möglichkeiten zur Produktivitätssteigerung und Qualitätsverbesserung ungenutzt läßt.

Der Forschungsansatz dieser Arbeit ist konstruktiv: In einem iterativen Prozeß wurden zunächst die theoretischen Grundlagen untersucht, danach Konzepte entworfen, in lauffähigen Prototypen realisiert und diese zum Experimentieren genutzt. Dabei stehen technische Fragen im Vordergrund, die beim kostengünstigen Bau flexibler PSEU beantwortet werden müssen; der Einsatz der Technologie zum Bau konkreter Werkzeuge und zur Unterstützung konkreter Prozesse ist Sache des Werkzeugentwicklers und des Prozeßmodellierers.

Der Ansatz geht von folgenden technischen Voraussetzungen aus [208]:

- Die CASE-Werkzeuge verwalten ihre Daten feingranular in einem OMS.
- Die CASE-Werkzeuge weisen die in früheren Arbeiten entwickelte OMS-orientierte Architektur auf, um die Dienste des OMS auch tatsächlich nutzen zu können.

6.1.1 Werkzeugbau

genform stellt einige Bausteine für PSEU zur Verfügung: Ein Framework mit Werkzeugkomponenten, aus denen CASE- und Prozeßwerkzeuge zusammengesetzt werden können, Mittel zur Spezifikation von Werkzeugen und Prozessen, Datenmodelle zur Verwaltung von Software-Dokumenten und Prozeßzuständen, sowie Meta-Werkzeuge zur Unterstützung des Werkzeugentwicklers. Die gebauten Werkzeuge erfüllen Qualitätsanforderungen hinsichtlich Komfort, Funktionsumfang und Geschwindigkeit – eine Voraussetzung dafür, überhaupt sinnvoll einsetzbar zu sein. Zusätzlich ist der Ansatz hinreichend flexibel, um (prinzipiell) den Bau aller Arten von *Upper-CASE*-Werkzeugen zu unterstützen – und dies zu geringeren Kosten, als bei einer herkömmlichen Programmierung unter Verwendung von Funktions- und Klassenbibliotheken. Durch den geringen Aufwand wird auch der rasche Bau von Prototypen ermöglicht, so daß Anforderungen an die Werkzeugunterstützung iterativ ermittelt und validiert werden können.

In *genform* werden Werkzeuge zum einen durch die Spezifikation der Werkzeugeigenschaften mit Werkzeugschemata, zum anderen durch die Implementierung werkzeugspezifischer Komponenten gebaut – somit wird eine einfache Anpassung und Erweiterung im ersten Fall mit einer hohen Flexibilität und Erweiterbarkeit im zweiten kombiniert.

Die Kosten spielen natürlich auch bei der Entwicklung des Werkzeug-Konstruktionsansatzes eine große Rolle und gehen letztendlich mit in die Werkzeugkosten ein. In *genform* werden die Kosten reduziert durch die Nutzung eines OMS, das die Daten der PSEU verwaltet und zahlreiche Dienste anbietet. Diese Dienste werden von den Werkzeugen direkt genutzt und müssen somit nicht in den Werkzeugkomponenten implementiert werden.

Zum anderen kann das Framework einfach um neue Komponenten erweitert werden – ohne daß dazu eine Spezifikationssprache, ein Generator oder Interpreter geändert werden müßte. Werden zunächst werkzeugspezifisch implementierte Komponenten verallgemeinert und in das Framework aufgenommen, so reift es parallel zur Werkzeugentwicklung. Zusätzlicher Aufwand entsteht hier nur bei der Verallgemeinerung der Komponenten, die wiederum die Einsetzbarkeit verbessert.

genform erweitert also frühere Ansätze [68, 69] sowohl unter technischen Gesichtspunkten, da es zusätzliche OMS-Dienste zur Realisierung der Werkzeugfunktionen nutzt und die verschiedenen Schichten in der Werkzeugarchitektur stärker isoliert; als auch konzeptionell, da die Eigenschaften von Werkzeugen durch Werkzeugschemata spezifiziert und die Werkzeuge zur Laufzeit aus Komponenten zusammengesetzt werden. Durch diese Möglichkeit zur Konfiguration wird auch die Adaptierung der Werkzeuge an die Anforderungen verschiedener Rollen oder Aufgaben im Software-Entwicklungsprozeß ermöglicht.

6.1.2 Prozeßunterstützung

genform unterstützt die Beschreibung von Software-Entwicklungsprozessen durch eine einfache Prozeß-Modellierungssprache. Diese Sprache wurde entwickelt, weil

– mit ihr spezifiziert werden kann, wie feingranulare Dokumente im Prozeß gehandhabt und mit welchen Werkzeugen sie bearbeitet werden sollen. Durch die Spezifikation von Werkzeugen mit Werkzeugschemata kann somit dem Bearbeiter einer Aufgabe die passende

Arbeitsumgebung zur Verfügung gestellt werden.
- die resultierenden Prozeßmodelle und Prozeßinstanzen im OMS verwaltet und mit *genform*-Werkzeugen bearbeitet werden können.
- sie einfach zu implementieren und einfach zu nutzen ist und zur Demonstration der entwickelten Konzepte ausreicht.

Im Vordergrund steht die Realisierung der Mechanismen zur Steuerung und Überwachung der Prozeßausführung, also der Prozeßmaschine und der Prozeßwerkzeuge: Auch diese Bausteine der PSEU müssen kostengünstig hergestellt und an die jeweilige Einsatzsituation angepaßt werden können. Somit liegt es nahe, die Ideen aus dem Werkzeugbau auf den Bau der gesamten PSEU zu übertragen, hier zu validieren und Synergieeffekte zu nutzen:

- Werkzeugkomponenten können in den Prozeßwerkzeugen wiederverwendet werden, wodurch der Implementierungsaufwand für die Bausteine der PSEU verringert wird.
- CASE- und Prozeßwerkzeuge können leichter integriert werden, weil sie das gemeinsame Repository zur Datenverwaltung nutzen und somit von CASE-Werkzeugen durchgeführte Änderungen bei der Steuerung des Prozesses berücksichtigt werden können.
- Die Benutzungsschnittstellen von CASE- und Prozeßwerkzeugen können einfach integriert werden, so daß die Belastung des Entwicklers durch zusätzliche Werkzeuge und GUI verringert wird.

Der komponentenbasierte Ansatz im Werkzeugbau findet seine Entsprechung im komponentenbasierten Bau der Prozeßmaschine: Jeder Aufgabe wird eine Komponente in der Prozeßmaschine zugeordnet, die für die Ausführung der Aufgabe verantwortlich ist – sie heißt Prozeßagent. Ein Prozeßagent wird durch das Prozeßmodell parametrisiert und kann weitere Komponenten laden. Das Verhalten der Prozeßmaschine wird also gesteuert durch das Prozeßmodell und die Implementierung der einzelnen Komponenten. Letztere stammen aus dem *genform*-Framework oder werden prozeßspezifisch angepaßt und erweitert.

6.2 Bewertung

Folgende Entwurfsentscheidungen wurden im Rahmen der Arbeit getroffen:

- Die aus *ToolFrame* bekannte Interpreter-Architektur wird so erweitert, daß auch die Eigenschaften und die Zusammensetzung von Werkzeugen spezifiziert und mit dem Datenmodell konsistent gehalten werden kann.
- Werkzeuge werden zur Laufzeit aus Komponenten zusammengesetzt.
- Das OMS wird auch zur Verwaltung des Prozeßzustands genutzt und die Datenmodelle für Dokumente und den Prozeßzustand werden integriert.
- Prozeßwerkzeuge werden mit dem gleichen Ansatz gebaut wie CASE-Werkzeuge und die Spezifikation der Werkzeuge aus dem Prozeßmodell generiert.
- Meta-Werkzeuge zur Unterstützung des Werkzeugentwicklers werden ebenfalls mit *genform* gebaut.

Der Bau von experimentellen Prototypen hat gezeigt, daß der Ansatz praktikabel ist: Es konnten mit sehr beschränkten Ressourcen funktionsfähige Werkzeuge gebaut und integriert werden. Eine Validierung des Ansatzes in der Praxis war nicht möglich, da die Werkzeuge und das OMS gegenüber kommerziellen Systemen einige Schwächen aufweisen und der Einsatz der Werkzeuge in einer Firma oder Organisation für diese einen nicht zu vernachlässigenden Umstellungsaufwand bedeutet hätte.

Übertragbarkeit

Ein Entwurfsziel von *genform* ist es, die Dienste des OMS möglichst gut auszunutzen, um so den Implementierungsaufwand für das Framework zu verringern. Daher stellt sich die Frage, wie aufwendig eine Übertragung von *genform* auf ein anderes Datenverwaltungssystem wäre, das die benötigten Dienste nicht zur Verfügung stellt. Außerdem ist zu klären, welche Technologien als Ersatz nutzbar sind. Ein vielversprechender Ansatz ist die Verwendung von XML-Dateien. Der entstehende Aufwand kann hier allerdings nur grob geschätzt werden – zur genaueren Bewertung müßten Werkzeugkomponenten experimentell realisiert werden. Bei der folgenden Betrachtung werden die verschiedenen Funktionsbereiche von H-PCTE unterschieden:

1. *Schemata und Sichten*: Die Schema-Interpretation erfordert Schnittstellen des Datenverwaltungssystems, über die Informationen über das Datenmodell des Werkzeugs abgefragt werden können. Außerdem müssen die Typinformationen um Werkzeugparameter erweitert werden können. Die Meta-Datenbank ist bereits in einer *genform*-Komponente gekapselt, die Typinformationen und Werkzeugparameter verwaltet und Schnittstellen zur Abfrage anbietet. Die darunterliegende Meta-Datenbank von H-PCTE könnte daher leicht ersetzt werden. Würden statt einer H-PCTE-Objektbank XML-Dateien verwendet, könnte zur Definition des Datenmodells eine der Schemasprachen für XML [221] verwendet werden. Bei *XML Schema* [109] werden die Typinformationen in XML-Dateien verwaltet, so daß auch die Erweiterung des Schemas um die Werkzeugparameter und die Abfrage des Schemas aus dem Werkzeug heraus möglich wäre.

   Sichten werden im Zusammenhang mit XML meist als Anfragen aufgefaßt, formuliert etwa mit *XSL* [56]. Abiteboul [1] berücksichtigt auch die Möglichkeit, Änderungen in Sichten durchzuführen, die dann in andere Sichten propagiert werden. Auf diese Weise könnte der Sichtenmechanismus von H-PCTE nachgebildet werden. Dazu müßte allerdings eine Komponente implementiert werden, die die Verwaltung der Sichten und die Änderungspropagation übernimmt. Es gibt noch kein standardisiertes API zur Änderung von XML-Daten, aber Vorschläge wie *XML:DB XUpdate* [220]. Mit *XUpdate* werden Änderungen deklarativ in XML-Syntax beschrieben. Um die Auswirkungen dieser Änderungen feststellen zu können, müssen die Beschreibungen also ausgewertet und in Änderungen der Daten in anderen Sichten umgesetzt werden. Die Realisierung einer solchen Komponente ist also aufwendig.
2. *Benachrichtigungen*: Der Benachrichtigungsmechanismus von H-PCTE informiert Werkzeuge (oder allgemein H-PCTE-Prozesse) über Änderungen, die parallel laufende Werkzeuge/Prozesse durchgeführt haben. Die Implementierung umfaßt also die Verwaltung von Notifizierern, die Überwachung von Änderungen und das Zustellen der Änderungsnachrichten. Wie oben beschrieben, ist der Benachrichtigungsmechanismus eng mit dem

Sichtenmechanismus verknüpft, da sich Änderungen auf Daten in verschiedenen Sichten auswirken können.

Existiert eine zentrale Komponente, die für die Datenverwaltung zuständig ist, scheint der Implementierungsaufwand für den Benachrichtigungsmechanismus überschaubar – allerdings müssen auch hier die im vorigen Punkt genannten Probleme bezüglich der Änderungsschnittstelle gelöst werden. Bei einer Implementierung in Java können Änderungsnachrichten per RMI-Aufruf übermittelt werden: Die aufgerufene Operation im Werkzeug aktualisiert dann den Zustand und die Darstellung des Werkzeugs.

Der *ActiveView*-Ansatz [1] implementiert die Änderungspropagation für XML-Dateien; hier können sogar Bedingungen angegeben werden, unter denen eine Änderungsnachricht zugestellt werden soll. Die Benachrichtigung übernehmen *view server*, die jeweils einer oder mehreren Anwendungen zugeordnet sind und auf einem zentralen *data server* arbeiten.

3. *Anfragesprache*: In *genform* wird die Anfragesprache *Ntt* [152] für Such- und Auswertungsfunktionen verwendet. Sie hilft, den Implementierungsaufwand zu verringern und in vielen Fällen die Geschwindigkeit der Anfrageverarbeitung zu erhöhen. Entscheidend für den Implementierungsaufwand ist, daß *Ntt* Mengen von Objektreferenzen zurückliefert. Über die Objektreferenzen wird auf die gefundenen Objekte zugegriffen – somit müssen keine neuen Datenstrukturen zur Verwaltung des Anfrageergebnisses in *genform* implementiert und die Werkzeuge daran angepaßt werden. Für XML-Dateien stehen zahlreiche Anfragesprachen zur Verfügung (etwa *XQuery* [50], *XQL* [283], *Lorel* [2]). Eine Anfrage liefert wiederum XML-Daten als Ergebnis; die Anfragesprachen sind also mit geringem Aufwand in einem Werkzeug nutzbar, das auf XML-Daten arbeitet.
4. *Transaktionen*: Transaktionen regeln in H-PCTE den parallelen Zugriff auf die Ressourcen in der Objektbank. Der Transaktionsmechanismus enthält feingranulare Sperren und ermöglicht das Rückgängigmachen und Wiederholen von Operationsfolgen (*undo/redo*) [270]. Die Implementierung von Transaktionen und Sperren ist mit einem erheblichen Aufwand verbunden. Bei einem Verzicht auf den Transaktionsmechanismus wären Konflikte zwischen parallel arbeitenden Werkzeugen möglich; außerdem müßte das *undo/redo* im Werkzeug implementiert werden. Da die Tätigkeiten der Benutzer durch das Prozeßmodell geregelt werden, sind Konflikte bei paralleler Bearbeitung in *genform*-Umgebungen weniger wahrscheinlich. Außerdem werden Benutzer durch den Benachrichtigungsmechanismus über die Änderungen anderer Benutzer informiert, so daß Konflikte unmittelbar sichtbar sind und nicht erst beim Speichern des Dokuments auftreten.

   Undo/redo-Funktionen, die auf einzelne Werkzeuge beschränkt sind, können mit überschaubarem Aufwand implementiert werden, da die Editierkommandos bereits als Werkzeugkomponenten realisiert sind: Jede Komponente müßte einen Eintrag in einen Kommando-Log schreiben und eine Funktion implementieren, die das Kommando rückgängig macht. Im Vergleich zur Nutzung des Transaktionsmechanismus von H-PCTE entstünde also zusätzlicher Aufwand bei der Implementierung von Werkzeugkommandos und eine mögliche Fehlerquelle, wenn die Log-Einträge oder die Funktion zum Rückgängigmachen des Kommandos nicht korrekt sind. In H-PCTE werden Abhängigkeiten zwischen Transaktionen anhand der durchgeführten OMS-Operationen ermittelt und beim *undo/redo* berücksichtigt. Dies ist beim Logging auf der Ebene von Benutzerkommandos nicht möglich, da deren Semantik von außen nicht zugänglich ist.
5. *Zugriffsrechte*: Die Rechteverwaltung in H-PCTE umfaßt die Verwaltung von Benutzern

und Benutzergruppen, die Zuordnung von ACL zu Objekten und die Prüfung der Rechte beim Zugriff auf die Objektbank. Das Nachimplementieren dieser Funktionen in *genform* wäre mit einem erheblichen Aufwand verbunden. Ein Verzicht auf Zugriffsrechte ist aber nicht möglich, da Zugriffsrechte die Editier- und Navigationsmöglichkeiten der Benutzer abhängig vom Prozeßzustand beeinflussen. Ein Kompromiß könnte darin bestehen, bei einer Speicherung in XML-Dateien auf die Rechteverwaltung des Betriebssystems zurückzugreifen, wie dies die meisten dateibasierten Werkzeuge tun. Nachteile sind die gröberen Rechtemodi und die Tatsache, daß allen Elementen innerhalb einer Datei die gleichen Rechte zugewiesen werden: Wird also jedes Dokument in einer Datei gespeichert, können keine unterschiedlichen Rechte an den verschiedenen Klassen innerhalb des Dokuments vergeben werden.

Für die Schema- und Sichtenverwaltung, für Anfragesprachen und den Benachrichtigungsmechanismus sind also Alternativen denkbar. Hier wäre zu prüfen, wie weit die genannten Technologien tatsächlich als Ersatz für die Dienste von H-PCTE nutzbar sind. Der Verzicht auf Transaktionen und Sperren hätte einen deutlich höheren Implementierungsaufwand für die *undo/redo*-Funktionen zur Folge und eine Einschränkung der Umgebungsfunktionen, da parallele Änderungen nicht synchronisiert würden. Ein Verzicht auf die Rechteverwaltung von H-PCTE bedeutet ebenfalls einen großen Implementierungsaufwand – oder alternativ eine starke Einschränkung der Umgebungsfunktionen, da Zugriffsrechte in *genform* zur feingranularen Steuerung der verfügbaren Kommandos und Darstellungen verwendet werden.

Praxisrelevanz

Bei der Übertragung des Ansatzes in die Praxis stellt sich das Problem, daß in einer Organisation meist schon Werkzeuge vorhanden sind, die zunächst ersetzt werden müßten. Zwar könnten vorhandene Werkzeuge nachträglich in die PSEU integriert werden, wie dies bei vielen Ansätzen möglich ist – allerdings gingen in diesem Fall die Vorteile der integrierten Werkzeuge verloren. Voraussetzung für den Einsatz wäre also ein ausreichender Leidensdruck aufgrund fehlender Werkzeugunterstützung für spezielle Methoden oder Qualitätsprobleme aufgrund nicht befolgter Vorschriften für die Prozeßdurchführung.

Ein weiteres Problem ist die Skalierbarkeit: Das zentrale Repository kann sich schnell als Flaschenhals erweisen; ein entfernter Zugriff ist nur bei ausreichender Bandbreite sinnvoll, da alle Aktionen im Werkzeug Zugriffe auf das Repository auslösen. Hier fehlen Mechanismen zur Versionierung, Verteilung und Replizierung von Daten, die aber außerhalb der Werkzeuge und des Werkzeug-Konstruktionsansatzes liegen.

6.3 Ausblick

Folgende Fragen könnten in anschließenden Arbeiten untersucht und beantwortet werden:

Validierung

Zu klären bleibt, welche Vorteile die Idee in der Praxis bringt, Werkzeuge an die konkreten Anforderungen anzupassen und in eine Beschreibung des Prozesses einzubetten:

- Wie groß sind die Kosten für die Anpassung der Werkzeuge und die Modellierung des Prozesses im Vergleich zu den erzielten Produktivitäts- und Qualitätsverbesserungen [303, 312]? Hier wirken sich die geringen Kosten für den Werkzeugbau und die hohe Flexibilität bei der Anpassung positiv aus, sowie die Tatsache, daß auch die Auswahl und Einführung kommerzieller Werkzeuge kostenintensiv sind.
- Müssen vorhandene Werkzeuge weiterhin genutzt werden? In diesem Fall müßte über eine nachträgliche Integration von Werkzeugen nachgedacht werden. In den *genform*-Werkzeugen ist bereits der Im- und Export von Daten im XML-Format vorgesehen, so daß der Datenaustausch mit anderen Werkzeugen grundsätzlich möglich wäre [156]. Eine andere Möglichkeit besteht darin, zu untersuchen, welche Erweiterungen an *genform* nötig wären, um die benötigten Werkzeugfunktionen realisieren zu können, und welchen Aufwand dies bedeuten würde.
- Ist die Prozeßunterstützung in der Praxis von Vorteil? Hierzu müßten die Gründe für das Scheitern von PSEU im praktischen Einsatz [123] ermittelt und geprüft werden, welche Vorteile *genform* bietet, die den Nutzen von PSEU erhöhen könnten. Zu berücksichtigen ist hier, daß die Werkzeuge an die konkreten Aufgaben eines Entwicklers angepaßt werden, somit die Komplexität der Werkzeuge reduziert wird. Durch die Integration von Werkzeugen und Prozeß entsteht dabei nur wenig Mehraufwand für den Entwickler: Mit den CASE-Werkzeugen durchgeführte Aktionen werden direkt bei der Prozeßkontrolle berücksichtigt. Auch kann die Prozeßunterstützung fallbezogen angepaßt werden – eher strenge Kontrolle bei Änderungsprozessen, größere Freiheit bei Analyse und Entwurf. Durch den Bau von Prototypen kann so versucht werden, eine passende Werkzeugunterstützung anzunähern.

Technische Umsetzung

Hier stellt sich die Frage, ob bessere Möglichkeiten zur technischen Umsetzungen von PSEU mit den beschriebenen Eigenschaften gefunden werden können: Das zentrale Repository schränkt die Skalierbarkeit des Ansatzes stark ein. Durch Optimierungen in H-PCTE und *genform* könnte das Problem allerdings gemildert werden. Eine alternative PSEU-Architektur, in der auch die Daten verteilt und somit von den einzelnen PSEU lokal bearbeitet werden können [98, 124], würde den potentiellen Flaschenhals beseitigen. Allerdings ist die Frage, wie feingranulare Daten versioniert und verteilt werden können, bisher noch nicht erschöpfend beantwortet worden [264].

Würden die Werkzeuge auf einer Komponentenarchitektur wie *Java Beans* basieren, könnten prinzipiell auch fremde Komponenten in die Werkzeuge integriert werden. Zu klären ist hier, wie die Interpretation der Werkzeugschemata mit einer solchen Komponentenarchitektur integriert werden kann, da die Eigenschaften und Schnittstellen bei Verwendung fremder Komponenten stärker variieren würden.

Literaturverzeichnis

[1] S. Abiteboul. On Views and XML. In ACM, editor, *Proc. 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 1999: Philadelphia, Pennsylvania, May 31–June 2, 1999*, pages 1–9, New York, NY 10036, USA, 1999. ACM Press.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[3] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A Query Language for Manipulating Object-oriented Databases. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 433–442, Amsterdam, The Netherlands, 22–25 Aug. 1989. Morgan Kaufmann.

[4] A. Alderson. Meta-CASE Technology. In A. Endres and H. Weber, editors, *Proceedings European Symposium on Software Development Environments and CASE Technology*, volume 509 of *Lecture Notes in Computer Science*, pages 81–91, Heidelberg, June 1991. Springer.

[5] A. Alderson, J. Cartmell, and A. Elliott. Toolbuilder: From CASE Tool Components to Method Engineering. In Gray et al. [134], pages 9–18.

[6] I. Alloui and F. Oquendo. To be consistent or not to be consistent in Cooperative software processes, that is the question. In *Proc. Int. Process Technology Workshop (IPTW'99)*, Villard de Lans, Sept. 1999.

[7] L. Alloui, S. Latrous, and F. Oquendo. A Multi-Agent Approach for Modelling, Enacting and Evolving Distributed Cooperative Software Processes. In Montangero [250], pages 225–235.

[8] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing Process-Centered Software Engineering Environments. *ACM Transactions on Software Engineering Methodology*, 6(3), July 1997.

[9] M. Amiour and J. Estublier. A Support for Communication in Software Processes. In *Proc.* 10th *Int. Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco Bay, USA, June 1998.

[10] W. Appelt. WWW Based Collaboration with the BSCW System. In *Proc. SOFSEM'99*, number 1725 in Lecture Notes in Computer Science, pages 66–78, Milovy, Czech Republic, Nov. 1999.

[11] S. Arbaoui, J. Lonchamp, and C. Montangero. The Human Dimension of the Software Process. In Derniame et al. [79], pages 165–199.

[12] S. Arbaoui and F. Oquendo. PEACE: Goal-Oriented Logic-Based Formalism for Process Modelling. In Nuseibeh et al. [262], pages 249–278.

[13] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti. A Survey and Assessment of Software Process Representation Formalisms. *Int. Journal on Software Engineering and Knowledge Engineering*, 3(3):401–426, 1993.

[14] M. Atkinson, P. Bailey, K. Chrisholm, W. Cockshott, and R. Morrison. An Approach to Persistent Programming. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 141–146. Morgan Kaufmann, 1990.

[15] J. H. August. *Joint Application Design: The Group Session Approach to System Design.* Prentice Hall/Yourdon Press, Englewood Cliffs, 1991.

[16] D. Avrilionis, P.-Y. Cunin, and C. Fernström. OPSIS: A View Mechanism for Software Processes which Supports their Evolution and Reuse. In *Proceedings of the 18th International Conference on Software Engineering*, pages 38–47. IEEE Computer Society Press, Mar. 1996.

[17] H. Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akadem. Verlag, Heidelberg, Berlin, 1998.

[18] H. Balzert, F. Hofmann, and C. Niemann. Vom Programmieren zum Generieren – Auf dem Weg zur automatisierten Anwendungsentwicklung. In *Beiträge der GI-Fachtagung Softwaretechnik 95*, pages 126–135, Braunschweig, Oct. 1995.

[19] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The architecture of SPADE-1 process-centered SEE. In Warboys [318], pages 15–30.

[20] S. Bandinelli, E. DiNitto, and A. Fuggetta. Supporting Cooperation in the SPADE-1 Environment. *IEEE Transactions on Software Engineering*, 22(12):841–865, Dec. 1996.

[21] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, Research Studies Press Advanced Software Development Series, pages 223–247. Research Studies Press Ltd., 1994.

[22] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, Sept. 1991.

[23] N. S. Bargouthi, W. Emmerich, W. Schäfer, and A. Skarra. Information Management in Process-Centered Software Engineering Environments. In A. Fuggetta and A. Wolf, editors, *Software Process*. John Wiley & Sons, 1996.

[24] V. R. Basili, L. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. Technical Report CS-TR-3443, University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, Univ. of Maryland, College Park, MD, May 1995.

[25] D. Batory. Intelligent Components and Software Generators. Technical Report CS-TR-97-06, University of Texas, Austin, Apr. 1, 1997.

[26] W. Becker, C. Burger, J. Klarmann, O. Kulendik, F. Schiele, and K. Schneider. Rechnerunterstützung für Interaktionen zwischen Menschen. *Informatik Spektrum*, 22:422–435, 1999.

[27] N. Belkhatir, J. Estublier, and W. L. Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. In B. Nuseibeh, A. Finkelstein, and J. Kramer, editors, *Software Process Modelling and Technology*, pages 103–129. John Wiley and Sons, 1994.

[28] R. Ben-Natan. *CORBA: A guide to CORBA*. McGraw-Hill, 1995.

[29] I. Z. Ben-Shaul and G. E. Kaiser. Process Evolution in the Marvel Environment. In W. Schäfer, editor, *Proc. 8^{th} Int. Software Process Workshop: State of the Practice in Process Technology*, pages 104–106, Wadern, Germany, Mar. 1993. Position paper.

[30] I. Z. Ben-Shaul and G. E. Kaiser. A Configuration Process for a Distributed Software Development Environment. In *2nd Int. Workshop on Configurable Distributed Systems*, pages 123–134, Pittsburgh PA, March 1994.

[31] I. Z. Ben-Shaul and G. E. Kaiser. An Interoperability Model for Process-Centered Software Engineering Environments and its Implementation in Oz. Technical Report CUCS-034-95, Columbia University Department of Computer Science, Dec. 1995.

[32] I. Z. Ben-Shaul and G. E. Kaiser. Integrating Groupware Activities into Workflow Management Systems. In *Proc. 7^{th} Israeli Conference on Computer Based Systems and Software Engineering*, pages 140–149, Tel Aviv, Israel, June 1996.

[33] P. A. Bernstein. Database System Support for Software Engineering- An Extended Abstract. In *Proc. 9^{th} Int. Conference on Software Engineering*, pages 166–178, Monterey, CA, Mar. 1987. IEEE, ACM, Computer Society Press.

[34] P. A. Bernstein and U. Dayal. An Overview of Repository Technology. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20^{th} Int. Conference on Very Large Data Bases*, pages 705–713, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[35] E. Bertino, editor. *Proc. 14^{th} European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *Lecture Notes in Computer Science*, Sophia Antipolis and Cannes, France, June 2000. Springer.

[36] B. Boehm. Get Ready for Agile Methods, with Care. *IEEE Computer*, 35(1):64–69, Jan. 2002.

[37] G. Bolcer and R. Taylor. Endeavors: A Process System Integration Infrastructure. In *Proc. of the 4^{th} Int. Conference on the Software Process*, pages 76–89. IEEE Computer Society Press, 1996. http://www.ics.uci.edu/pub/endeavors/.

[38] L. Bompani, P. Ciancarini, and F. Vitali. Software Engineering on the Internet: A Roadmap. In Finkelstein [114], pages 303–318.

[39] S. Brinkkemper. Method Engineering: Engineering of Information Systems Development Methods and Tools. *Information and Software Technology*, 38(4):275–280, 1996.

[40] S. Brinkkemper, K. Lyytinen, and R. J. Weike. *Method Engineering*. Chapman & Hall, 1996.

[41] S. Brodsky. XMI Opens Application Interchange. White paper, IBM, Mar. 1999. `www-4.ibm.com/software/ad/standards/xmiwhite0399.pdf`.

[42] A. W. Brown and J. A. McDermid. Learning from IPSE's Mistakes. *IEEE Software*, 9(2):23–28, Mar. 1992.

[43] T. Bruckhaus. TIM: A Tool Insertion Method. In *Proc. IBM Centre for Advanced Studies (CAS) Conference (CASCON'94)*. IBM Canada Ltd. and The National Research Council of Canada, 1994.

[44] J. Bubenko. Selecting a Strategy for Computer-Aided Software Engineering (CASE). SYSLAB Report 59, SYSLAB, University of Stockholm, Sweden, 1988.

[45] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, Oct. 1991.

[46] M. R. Cagan. The HP SoftBench Environment Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[47] G. Canals, N. Boudjlida, J. C. Derniame, C. Godart, and J. Lonchamp. ALF: A Framework for Building Process-Centred Software Engineering Environments. In Nuseibeh et al. [262], pages 153–185.

[48] G. Canals, F. Charoy, C. Godart, and P. Molli. P-RooT & COO : Building a Cooperative Software Development Environment. In Proc. SEE'95 [279].

[49] CDIF Framework for Modelling and Extensibility. Technical Report EIA/IS-107, Electronic Industries Association, 1994.

[50] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu (Eds). XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. `http://www.w3.org/TR/2001/WD-xquery-20010607/`.

[51] A. Chaudhri and P. Osmon. A Comparative Evaluation of the Major Commercial Object and Object-Relational DBMSs: GemStone, O2, Objectivity/DB, ObjectStore, VERSANT ODBMS, Illustra, Odapter and UniSQL. Technical report, Computer Science Department, City University, London, Aug. 1996.

[52] P. Chen. The Entity-Relationship Model – toward a unified View of Data. In *ACM Transactions on Database Systems*, volume 1, 1976.

[53] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[54] A. Christie. Process-Centered Development Environments: An Exploration of Issues. Technical Report SEI-93-TR-4, Software Engineering Institute, Pittsburgh, Pa, June 1993.

[55] A. M. Christie, A. N. Earl, M. I. Kellner, and W. E. Riddle. A Reference Model for Process Technology. In Montangero [250], pages 3–17.

[56] J. Clark and S. Deach. Extensible Stylesheet Language (XSL), Version 1.0. World Wide Web Consortium Working Draft. `http://www.w3.org/TR/WD-xsl`, Aug 18, 1998.

[57] M. Clauß. Modellierung von Variabilitäten in UML. In *Proc. Informatiktage 2001*, Bad Schussenried, Nov. 2001. Gesellschaft für Informatik e.V.

[58] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Prentice Hall, 1991.

[59] C. Collet, P. Habraken, T. Coupaye, and M. Adiba. Active Rules for the Software Engineering platform GOODSTEP. In *Proc.* 2^{ndt} *Int. Workshop on Database and Software Engineering*, pages 165–180, May 1994.

[60] R. Conradi, C. Fernström, and A. Fuggetta. Concepts of Evolving Software Processes. In Nuseibeh et al. [262], pages 9–31.

[61] R. Conradi, M. Hagaseth, J. O. Larsen, M. N. Nguyen, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-Oriented Cooperative Process Modeling. In B. Nuseibeh, A. Finkelstein, and J. Kramer, editors, *Software Process Modelling and Technology*, pages 33–70. John Wiley and Sons, 1994.

[62] R. Conradi and M. L. Jaccheri. Process Modelling Languages. In Derniame et al. [79], pages 27–52.

[63] R. Conradi and C. Liu. Process Modelling Languages: One or Many? In Schäfer [288], pages 98–118.

[64] R. Conradi, E. Osjord, P. H. Westby, and C. Liu. Initial Software Process Management in EPOS. *Software Engineering Journal*, 6(5):275–284, Sept. 1991.

[65] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[66] B. Curtis, M. Kellner, and J. Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, Sept. 1992.

[67] J. L. Cybulski and K. Reed. A Hypertext-Based Software-Engineering Environment. *IEEE Software*, 9(2):62–68, Mar. 1992.

[68] D. Däberitz. *Der Bau von Software-Entwicklungsumgebungen mit Hilfe von Nicht-Standard-Datenbanken*. Shaker Verlag, Aachen, 1997.

[69] D. Däberitz and U. Kelter. Rapid Prototyping of Graphical Editors in an Open SDE. In Proc. SEE'95 [279], pages 61–72.

[70] A. Dahanayake. *Computer-Aided Method Engineering: Designing CASE Repositories for the* 21^{st} *Century*. Idea Group Publishing, Hershey, London, 2001.

[71] A. Dahanayake and G. Florijn. Evaluation of Object Oriented Database Support for Software Engineering Environments. In Proc. SEE'95 [279], pages 11–20.

[72] S. Dami, J. Estublier, and M. Amiour. APEL: A Graphical Yet Executable Formalism for Process Modeling. *Automated Software Engineering (ASE)*, March 1997.

[73] C. H. Damm, K. M. Hansen, M. Thomsen, and M. Tyrsted. Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools. In Bertino [35], pages 27–43.

[74] W. Deiters and V. Gruhn. Managing Software Processes in the Environment MELMAC. In *Proc. of the Fourth ACM SIGSOFT '90 Symposium on Software Development Environments*, pages 193–205, Dec. 1990. Published as SIGSOFT Software Engineering Notes, volume 15, number 6.

[75] B. Dellen and F. Maurer. Integrating Planning and Execution in Software Development Processes. In *Infrastructure for Collaborative Enterprises: Proc. of the Workshop on Enabling Technology: Infrastructure for Collaborative Enterprises (WET ICE 96), June 19–21, Stanford University, California, USA*, 1996.

[76] F. Demuth and J. Dierks. Entscheidungskriterien zur Auswahl von 4GL-Systemen. *iX–Multiuser Multitasking Magazin*, Jan. 1995.

[77] J. Derniame, editor. *Software Process Technology: Proceedings Second European Workshop, EWSPT '92, Trondheim, Norway, September 1992*, volume 635 of *Lecture Notes in Computer Science*, Heidelberg, Sept. 1992. Springer–Verlag.

[78] J. Derniame and V. Gruhn. Development of Process-Centered IPSEs in the ALF Project. *Journal of Systems Integration*, (4):127–150, 1994.

[79] J. Derniame, B. A. Kaba, and D. Wastell, editors. *Software process: Principles, Methodology, and Technology*, Berlin, 1999. Springer.

[80] J. C. Derniame. Life Cycle Process Support in PCIS. In T. Lindquist and H. Kaehnemann, editors, *Proc. PCTE 1994 Conference*, pages 97–110. PIMB Association, Nov. 1994.

[81] P. Dewan and J. Riedl. Toward Computer-Supported Concurrent Software Engineering. *Computer*, 26(1):17–27, Jan. 1993.

[82] G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka. Business Process Modeling in the Workflow Management Environment LEU. In P. Loucopoulos, editor, *Proc. of the* 13^{th} *Int. Conference on the Entity-Relationship Approach*, volume 881 of *Lecture Notes in Computer Science*, pages 46–63, Manchester, UK, Dec. 1994. Springer.

[83] R. Dirckze, D. Baisley, and S. Iyengar. XMI - A Model Driven XML Interchange Format. In SEF2000 [292].

[84] K. R. Dittrich and R. A. Lorie. Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, 14(4):429–437, Apr. 1988.

[85] K. R. Dittrich, D. Tombros, and A. Geppert. Databases in Software Engineering: A Roadmap. In Finkelstein [114], pages 25–34.

[86] J. Doppke, D. Heimbigner, and A. Wolf. Software Process Modeling and Execution within Virtual Environments. *ACM Transactions on Software Engineering and Methodology*, 7(1):1–40, Jan. 1998.

[87] S. E. Dossick and G. E. Kaiser. CHIME: A Metadata-Based Distributed Software Development Environment. In *Joint* 7^{th} *European Software Engineering Conference and* 7^{th} *ACM SIGSOFT Int. Symposium on the Foundations of Software Engineering*, September 1999.

[88] M. Dowson. ISTAR – An Integrated Project Support Environment. *SIGPLAN Notices*, 22(1):27–33, Jan. 1987.

[89] M. Dowson and C. Fernström. Towards Requirements for Enactment Mechanisms. In Warboys [318], pages 90–106.

[90] J. E. Dutton. Commonsense Approach to Process Modeling. *IEEE Software*, 10(4):56–64, July 1993.

[91] J. Ebert. MetaCASE: Generierung und Anpassung von CASE-Werkzeugen. In W. Gens, editor, *3. Fachkongress Smalltalk und Java in Industrie und Ausbildung (STJA'97), Technische Universität Ilmenau*, 1997.

[92] J. Ebert and C. Lewerentz, editors. *Proceedings of the* 8^{th} *Software Engineering Environments Conference*, Cottbus, Germany, 8–9 Apr. 1997. IEEE.

[93] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in Practice: a Case for KOGGE. In J. A. P. A. Olive, editor, *Advanced Information Systems Engineering, Proc.* 9^{th} *Int. Conference on Advanced Information Systems Engineering (CAiSE'97)*, volume 1250 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 1997.

[94] J. Ebert, A. Winter, P. Dahm, R. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In *Proc.* 15^{th} *Int. Conference on Conceptual Modeling (ER'96)*, volume 1157 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1996.

[95] C. Ellis, S. Gibbs, and G. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 34(1):38–58, Jan. 1991.

[96] W. Emmerich. *Tool Construction for Process-Centered Software Development Environments.* Universität-Gesamthochschule Paderborn, Paderborn, 1995.

[97] W. Emmerich. Tool Specification with GTSL. In *Proc. of the* 8^{th} *Int. Workshop on Software Specification and Design*, pages 26–35, Schloß Velen, Germany, 1996. IEEE Computer Society Press.

[98] W. Emmerich. CORBA and ODBMSs in Viewpoint Development Environment Architectures. In *Proc.* 4^{th} *Int. Conference on Object-Oriented Information Systems*, pages 347–360. Springer Verlag, 1997.

[99] W. Emmerich, S. Bandinelli, L. Lavazza, and J. Arlow. Fine-grained Process Modelling: An Experiment at British Airways. In *Proc.* 4^{th} *Int. Conference on the Software Process (ICSP '96)*. Institute of Electrical and Electronics Engineers, Inc., 1996.

[100] W. Emmerich, G. Ferran, F. Ferrandina, A. Fuggetta, C. Ghezzi, S. Lautemann, L. Lavazza, J. Madec, M. Phoenix, S. Sachweh, W. Schäfer, C. S. dos Santos, G. Tigg, and R. Zicari. The GOODSTEP Project: General Object-Oriented Database for Software Engineering Processes. In *Proc. of the* 1^{st} *Asian Pacific Software Engineering Conference*, pages 10–19, Tokyo, Japan, 1994. IEEE Computer Society Press.

[101] W. Emmerich and A. Finkelstein. Do Process-Centred Environments Deserve Process-Centred Tools? In C. Montangero, editor, *Proc. of the* 5^{th} *European Workshop on Software Process Technology (EWSPT'96)*, volume 1149 of *Lecture Notes in Computer Science*, pages 75–81. Springer, 1996.

[102] W. Emmerich, P. Kroha, and W. Schäfer. Object-Oriented Database Management Systems for Construction of CASE Environments. *Lecture Notes in Computer Science*, 720:631–642, 1993.

[103] W. Emmerich and W. Schäfer. Groupie – An Environment supporting Group-Oriented Architecture Development. Technical Report 2, ESPRIT-III Project GOODSTEP (6115), Feb. 1994.

[104] W. Emmerich, W. Schäfer, and J. Welsh. Suitable Databases for Process Centered Environments do not yet exist. In Derniame [77], pages 94–98.

[105] W. Emmerich, W. Schäfer, and J. Welsh. Databases for Software Engineering Environments: The Goal Has Not Yet Been Attained. In I. Sommerville and M. Paul, editors, *Proc. European Software Engineering Conference (ESEC'93)*, volume 717 of *Lecture Notes in Computer Science*, pages 145–162. Springer-Verlag, 1993.

[106] G. Engels, L. Groenewegen, and G. Kappel. Object-oriented Specification of Coordinated Collaboration. In N. Terashima and E. Altman, editors, *Proc. IFIP World Conference on IT Tools*, pages 437–449, London, New York, Sept. 1996. Chapman & Hall.

[107] J. Estublier. Work Space Management in Software Engineering Environments. In I. Sommerville, editor, *Software configuration management: ICSE'96 SCM-6 Workshop*, volume 1167 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 1996.

[108] J. Estublier and R. Casallas. The Adele Configuration Manager. In W. Tichy, editor, *Configuration Management*, pages 99–133. John Wiley and Sons, 1994.

[109] D. C. Fallside (Eds). XML Schema Part 0: Primer. W3C Recommendation, May 2001. `http://www.w3.org/TR/xmlschema-0`.

[110] R. Ferguson, N. Parrington, P. Dunne, J. Archibald, and J. Thompson. MetaMOOSE – an Object-Oriented Framework for the construction of CASE tools. In Gray et al. [134].

[111] C. Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *Proc. of the* 2^{nd} *Int. Conference on the Software Process*, pages 12–26, Los Alamitos, CA., 1993. IEEE Computer Society Press.

[112] P. Findeisen. The Graphical Extension for the EARA Model. Technical report, Department of Computing Science, University of Alberta, 1993.

[113] P. Findeisen. The EARA Model for Metaview. Technical report, Department of Computing Science, University of Alberta, 1994.

[114] A. Finkelstein, editor. *Proceedings of the* 22^{nd} *Int. Conference on Software Engineering (ICSE'00) – Track "The Future of Software Engineering"*, NY, June 4–11 2000. ACM Press.

[115] A. Finkelstein, J. Kramer, and M. Goedicke. Viewpoint Oriented Software Development. In *Proc.* 3^{rd} *Int. Workshop on Software Engineering and its Applications*, Toulouse, Dec. 1990.

[116] D. Fischer and H. Utermann. SurfBorD – Systemunabhängig realisierte flexible Bedienoberfläche für relationale Datenbanken. In *Proc. Java-Informationstage 1998 (JIT'98)*, pages 87–98. Springer, 1998.

[117] M. Foegen and J. Battenfeld. Die Rolle der Architektur in der Anwendungsentwicklung. *Informatik Spektrum*, 24(5):290–304, Oct. 2001. in German.

[118] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. In Bertino [35], pages 63–82.

[119] M. Fowler. The New Methodology. `http://www.martinfowler.com/articles/NewMethodology.htm`, 2001.

[120] P. Fradet, D. L. Métayer, and M. Périn. Consistency Checking for Multiple View Software Architectures. In O. Nierstrasz and M. Lemoine, editors, *Proc.* 7^{th} *European Engineering Conference and* 7^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engeneering (ESEC/FSE'99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428, Berlin, Heidelberg, Sept. 6–10 1999. Springer.

[121] A. Fuggetta. A Classification of CASE Technology. *IEEE Computer*, 26(12):25–38, Dec. 1993.

[122] A. Fuggetta. Functionality and Architecture of PSEEs. *Information and Software Technology*, 38(4):289–293, 1996.

[123] A. Fuggetta. Software Process: A Roadmap. In Finkelstein [114], pages 25–34.

[124] A. Fuggetta, C. Godart, and J. Jahnke. Architectural Views and Alternatives. In Derniame et al. [79], pages 96–116.

[125] M. Fugini, O. Nierstrasz, and B. Pernici. Application Development through Reuse: the Ithaca Tools Environment. In *ACM SIGOIS Bulletin*, volume 13:2, pages 38–47, Aug. 1992.

[126] P. H. F. Gail E. Kaiser and S. S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[127] E. Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++.* Springer, Heidelberg, 1992.

[128] P. K. Garg, P. Mi, T. Pham, W. Scacchi, and G. Thunquest. The SMART Approach for Software Process Engineering. In B. Fadini, editor, *Proc.* 16^{th} *Int. Conference on Software Engineering*, pages 341–352, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[129] M. Gille. *Diagramm-Editoren: Generierung aus objektorientierten Modellinformationen.* Spektrum Akadem. Verlag, Heidelberg, Berlin, 1999.

[130] C. Godart, G. Canals, F. Charoy, and P. Molli. About some Relationships Between Configuration Management, Software Process and Cooperative Work: The COO Environment. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 173–178. Springer-Verlag, Oct. 1995.

[131] C. Godart and D. Dietrich. Stepwise Specification of Interactive Processes in COO. In Schäfer [288], pages 220–239.

[132] S. Goldmann, J. Münch, and H. Holz. MILOS: A Model of Interleaved Planning, Scheduling, and Enactment. In *Proc.* 2^{nd} *Workshop on Software over the Internet*, May 1999.

[133] I. Gorton and S. Motwani. Issues in Co-Operative Software Engineering Using Globally Distributed Teams. *Information and Software Technology*, 38(10):647–655, Oct. 1996.

[134] J. Gray, J. Harvey, A. Liu, and L. Scott, editors. *Proc.* 1^{st} *Int. Symposium on Constructing Software Engineering Tools (COSET'99)*, Los Angeles, CA, May 1999.

[135] F. Griffel. *Componentware – Konzepte und Techniken eines Softwareparadigmas.* dpunkt Verlag für digitale Technologie, Heidelberg, 1998.

[136] B. Gronemann, G. Joeris, S. Scheil, M. Steinfort, and H. Wache. Supporting Cross-Organizational Engineering Processes by Distributed Collaborative Workflow Management – The MOKASSIN Approach. In *Proc. 2^{nd} Symposium on Concurrent Mulitdisciplinary Engineering (CME'99) / 3^{rd} Int. Conf. on Global Engineering Networking (GEN'99)*, Bremen, Germany, Sept. 1999.

[137] J. Grudin. CSCW Introduction. *Communications of the ACM*, 34(12), Dec. 1991.

[138] V. Gruhn. Configuration Management Support for Software Process Models. In C. Montangero, editor, *Proc. 5^{th} European Workshop on Software Process Technology*, pages 132–136. Lecture Notes in Computer Science 1149, Springer, Oct. 1996.

[139] V. Gruhn. Generierung und individuelle Anpassung von Dialogen auf Basis von Objektmodellen. In *Proc. GI-Fachtagung Softwaretechnik 96*, pages 129–139, Koblenz, 12.–13.Sept. 1996.

[140] V. Gruhn, editor. *Proc. 6^{th} European Workshop on Software Process Technology (EWSPT'98)*, volume 1487 of *Lecture Notes in Computer Science*, Weybridge, UK, Sept. 1998. Springer.

[141] V. Gruhn and R. Jegelka. An Evaluation of FUNSOFT Nets. In Derniame [77], pages 196–214.

[142] V. Gruhn, C. Pahl, and M. Wever. Data Model Evolution as a Business Process Management. In *Proc. 14^{th} Int. Conference on Object-Oriented and Entity-Relationship Modelling*, volume 1021 of *Lecture Notes in Computer Science*, pages 270–281. Springer, Dec. 1995.

[143] J. Grundy, J. Hosking, and W. Mugridge. Low-level and High-level CSCW Support in the Serendipity Process Modelling Environment. In J. Grundy and M. Apperley, editors, *Proceedings of OZCHI96*, Hamilton, New Zealand, Nov. 1996. IEEE CS Press.

[144] J. Grundy, W. Mugridge, and J. Hosking. A Java-based Componentware Toolkit for Constructing Multi-view Editing Systems. In *Proc 2^{nd} Component Users' Conference (CUC'97)*, Munich, July 1997. SIGS Books.

[145] J. Grundy, W. Mugridge, and J. Hosking. Constructing Component-based Software Engineering Environments: Issues and Experiences. *Information and Software Technology – Special Issue on Constructing Software Engineering Tools*, 42(2):103–114, 2000.

[146] J. C. Grundy, M. Apperley, J. G. Hosking, and W. B. Mugridge. A Decentralized Architecture for Software Process Modeling and Enactment. *IEEE Internet Computing*, pages 53–62, Sept./Oct. 1998.

[147] J. C. Grundy and J. G. Hosking. Constructing Multi-View Editing Environments using MViews. In E. P. Glinert and K. A. Olsen, editors, *Proc. IEEE Symposium on Visual Languages*, pages 220–224. IEEE Press, 24–27 Aug. 1993.

[148] J. C. Grundy and J. G. Hosking. Serendipity: Integrated Environment Support for Process Modelling, Enactment and Work Coordination. *Automated Software Engineering*, 5(1):27–60, 1998.

[149] J. C. Grundy, W. B. Mugridge, and J. G. Hosking. Constructing Component-based Software Engineering Environments: Issues and Experiences. In Gray et al. [134].

[150] J. C. Grundy and J. R. Venable. Towards an Integrated Environment for Method Engineering. In *Method Engineering* [40], pages 45–62.

[151] B. Gulla. A Browser for a Versioned Entity-Relationship Database. In *Proc. Int. Workshop on Interfaces to Database Systems (IDS'92)*, Glasgow, Scotland, July 1992.

[152] O. Haase. *Ntt – a Set-Oriented Algebraic Query Language for PCTE.* Internes Memorandum 95/5, Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, May 1995.

[153] O. Haase. *Mengenorientierte Anfragen auf partiell zugreifbaren Datenbanken.* Shaker Verlag, Aachen, 1998.

[154] D. Harel and E. Gery. Executable Object Modeling with Statecharts. In *Proc. 18^{th} Int. Conference on Software Engineering*, pages 246–257, Berlin - Heidelberg - New York, Mar. 1996. Springer.

[155] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, Apr. 1990.

[156] W. Harrison, H. Ossher, and P. Tarr. Software Engineering Tools and Environments: A Roadmap. In Finkelstein [114], pages 261–277.

[157] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. DYNAMITE: Dynamic Task Nets for Software Process Management. In 18^{th} *Int. Conference on Software Engineering*, pages 331–341, Berlin - Heidelberg - New York, Mar. 1996. Springer.

[158] P. Heimann, C. Krapp, B. Westfechtel, and G. Joeris. Graph-Based Software Process Management. *Int. Journal on Software Engineering and Knowledge Engineering*, 7(4):431–456, 1997.

[159] D. Heimbigner. Proscription versus Prescription in Process-Centered Environments. In T. Katayama, editor, *Proc.* 6^{th} *Int. Software Process Workshop*, pages 99–102, Hakodate, Japan, Oct. 1990.

[160] D. Heimbigner. Experiences With an Object Manager for a Process-Centered Environment. Technical Report CU-CS-484-91, University of Colorado at Boulder, Boulder, CO 80309, 1991.

[161] D. Heimbigner. The ProcessWall: A Process State Server Approach to Process Programming. In *Proc. of the* 5^{th} *ACM SIGSOFT Symposium on Software Development Environments*, pages 159–168, Dec. 1992. Published as SIGSOFT Software Engineering Notes, Volume 17, Number 5.

[162] G. Heineman, J. Botsford, G. Caldiera, M. K. G.E. Kaiser, and N. Madhavji. Emerging Technologies that Support a Software Process Life Cycle. *IBM Systems Journal*, 33(3):501–529, 1994.

[163] J. C. Henderson and J. G. Cooprider. Dimensions of I/S Planning and Design Aids: A Functional Model of CASE Technology. *Information Systems Research*, 1(3):227–254, Jan. 1990.

[164] A. Henrich. P–OQL: An OQL-Oriented Query Language for PCTE. In Proc. SEE'95 [279], pages 48–60.

[165] A. Henrich. Document Retrieval Facilities for Repository-Based System Development Environments. In *Proc. ACM SIGIR Int. Conf. on Research and Development in Information Retrieval*, Zürich, Switzerland, 18.–22. Aug. 1996.

[166] A. Henrich and D. Däberitz. Using a Query Language to State Consistency Constraints for Repositories. In *Proc.* 7^{th} *Int. Conference and Workshop on Database and Expert Systems Applications*, Switzerland, 9.–13. Sept. 1996.

[167] A. Henrich and U. Kelter. Integration von Zugriffsparadigmen in einem Repository. In *Proc. GI-Jahrestagung '96*, Klagenfurt, 1996.

[168] J. Herbsleb, D. Zubrow, D. Goldenson, W. Hayes, and M. Paulk. Software Quality and the Capability Maturity Model. *Communications of the ACM*, 40(6):31–40, June 1997.

[169] A. Hess and H. Schulz. SHORE: An Example for an Exchange Format Based on XML. In SEF2000 [292].

[170] G. J. Hidding. Reinventing methodology: Who reads it and why? *Communications of the ACM*, 40(11):102–109, Nov. 1997.

[171] A. S. Hitomi and D. Le. Endeavors and Component Reuse in Web-Driven Process Workflow. In *Proceedings of the California Software Symposium*, Oct. 1998.

[172] W. S. Humphrey and M. I. Kellner. Software Process Modeling: Principles of Entity Process Models. In *Proc. of the 11th Int. Conference on Software Engineering*, pages 331–342, May 1989.

[173] J. Iivari. Why are CASE Tools not used? *Communications of the ACM*, 39(10):94–103, Oct. 1996.

[174] ISO. *Portable Common Tool Environment — Abstract Specification / C Bindings / Ada Bindings (Standards ECMA-149 / -158 / -165, 3rd Edition, and ISO IS 13719-1 / -2 / -3).* ISO, 1994.

[175] ISO. *Portable Common Tool Environment – Object Oriented Extensions — Abstract Specification / C Bindings / Ada Bindings (Draft Standard).* ISO, Sept. 1995.

[176] D. Jäger. Generating Tools from Graph-Based Specifications. In Gray et al. [134], pages 97–107.

[177] D. Jäger. Modelling Management and Coordination in Development Processes. In R. Conradi, editor, *Proc.* 7^{th} *European Workshop on Software Process Technology (EWSPT 2000)*, volume 1780 of *Lecture Notes in Computer Science*, pages 109–114, Heidelberg, 2000. Springer.

[178] D. Jäger, A. Schleicher, and B. Westfechtel. Using UML for Software Process Modeling. In O. Nierstrasz and M. Lemoine, editors, *Proc. 8^{th} European Software Engineering Conference (ESEC/FSE'99)*, volume 1687 of *Lecture Notes in Computer Science*, pages 91–108. Springer-Verlag / ACM Press, 1999.

[179] D. Jäger, A. Schleicher, and B. Westfechtel. AHEAD: A Graph-Based System for Modeling and Managing Development Processes. In M. Nagl and A. Schürr, editors, *Proc. Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, number 1779 in Lecture Notes in Computer Science, Castle Rolduc, 2000. Springer-Verlag.

[180] M. Jarke, K. Pohl, K. Weidenhaupt, K. Lyytinen, P. Marttiin, J.-P. Tolvanen, and M. Papazoglou. Meta Modelling: A Formal Basis for Interoperability and Adaptability. In B. Krämer and M. Papazoglou, editors, *Information Systems Interoperability*, pages 229–263. John Wiley Research Science Press, 1998.

[181] M. G. Jenner. *Software quality management and ISO 9001: how to make them work for you.* Wiley, New York, 1995.

[182] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based Business Process Management. *Int. Journal of Cooperative Information Systems*, 5(2&3):105–130, 1996.

[183] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, and B. Odgers. Autonomous Agents for Business Process Management. *Int. Journal of Applied Artificial Intelligence*, 14(2):145–189, 2000.

[184] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, B. Odgers, and J. L. Alty. Implementing a Business Process Management System using ADEPT: A Real-World Case Study. *Int. Journal of Applied Artificial Intelligence*, 14(5):421–465, 2000.

[185] N. R. Jennings, P. Faratin, T. J. Norman, P. O'Brien, M. E. Wiegand, C. Voudouris, J. L. Alty, T. Miah, and E. H. Mamdani. ADEPT: Managing Business Processes using Intelligent Agents. In *Proc. BCS Expert Systems 96 Conference (ISIP Track)*, pages 5–23, Cambridge, UK, 1996.

[186] N. R. Jennings and M. J. Wooldridge. Software Agents. *IEE Review*, pages 17–20, Jan. 1996.

[187] M. Jeusfeld, M. Jarke, H. Nissen, and M. Staudt. ConceptBase – Managing Conceptual Models about Information Systems. In P. Bernus, K. Mertins, and G. Schmidt, editors, *Handbook on Architectures of Information Systems*. Springer-Verlag, 1998.

[188] J.Lonchamp and B.Denis. Fine-grained Process Modelling for Collaborative Work Support: Experiences with CPCE . In *Proc. 8^{th} euroGDSS Workshop*, 1997.

[189] G. Joeris. Change Management Needs Integrated Process and Configuration Management. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 125–141. Springer / ACM Press, 1997.

[190] G. Joeris. Decentralized and Flexible Workflow Enactment Based On Task Coordination Agents. In *Proc. of the 2^{nd} Int. Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2000 @ CAiSE*00)*, pages 41–62, Stockholm, Sweden, 2000.

[191] G. Joeris and O. Herzog. Towards Flexible and High-Level Modeling and Enacting of Processes. In M. Jarke and A. Oberweis, editors, *Proc. 11^{th} Int. Conference on Advanced Information Systems Engineering (CAiSE '99)*, volume 1626 of *LNCS*, pages 88–102, Berlin, Germany, 1999. Springer.

[192] I. U. Jürgen Ebert, Roger Süttenbach. JKogge: a Component-Based Approach for Tools in the Internet. In *Proc. STJA '99*, Erfurt, September 1999.

[193] G. Junkermann. A Dedicated Process Design Language based on EER-models, Statecharts and Tables. In *Proc. 7^{th} Int. Conf. on Software Engineering and Knowledge Engineering*, pages 487–496. Knowledge Systems Institute, 1995.

[194] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In Nuseibeh et al. [262], pages 103–129.

[195] A. S. Karrer and W. Scacchi. Meta-Environments for Software Production. *International Journal of Software Engineering and Knowledge Engineering*, 3(1):139–162, 1993.

[196] M. H. Kay, P. J. Rivett, and T. J. Walters. The Raleigh Activity Model: Integrating Versions, Concurrency, and Access Control. In P. M. D. Gray and R. J. Lucas, editors, *Proc.* 10^{th} *British National Conference on Databases (BNCOD'92)*, volume 618 of *LNCS*, pages 175–191, Berlin,, July 1992. Springer.

[197] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proc.* 8^{th} *Int. Conference on Advanced Information Systems Engineering (CAiSE'96)*, volume 1080 of *Lecture Notes in Computer Science*, pages 1–21, Heraklion, Crete, Greece, May 1996. Springer-Verlag.

[198] U. Kelter. Gruppen-Transaktionen vs. gruppenorientierte Zugriffsrechte. In M. Paul, editor, *Proc. 19. Jahrestagung der Gesellschaft für Informatik*, volume 1 of *Informatik-Fachberichte*, pages 287–300. Springer, 1989.

[199] U. Kelter. Pcte. *Informatik Spektrum*, 12:165–166, 1989.

[200] U. Kelter. Group-Oriented Discretionary Access Controls for Distributed Structurally Object-Oriented Database Systems. In *Proc. European Symposium on Research in Computer Security*, Toulouse, 1990.

[201] U. Kelter. H-PCTE – a High Performance Object Management System for System Development Environments. In *Proc.* 16^{th} *Annual Int. Computer Software and Application Conference (COMPSAC'92)*, pages 45–50, Chicago, Illinois, Sept. 1992. IEEE Computer Society Press.

[202] U. Kelter. Integrationsrahmen für Software-Entwicklungsumgebungen. *Informatik Spektrum*, (16):281–285, 1993.

[203] U. Kelter. *Grundlegende Konzepte des Datenbankmodells von PCTE.* Lehrmodul zur Vorlesung Datenbanksysteme, Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 1999.

[204] U. Kelter and D. Däberitz. An Assessment of Non-Standard DBMSs for CASE Environments. In *Proc. Int. Conference on Extending Database Technology*, volume 1057 of *Lecture Notes in Computer Science*, pages 25–29. Springer, Mar. 1996.

[205] U. Kelter, D. Däberitz, and D. Platz. Integration of DBMSs and Other Framework Components. In *Workshop on Research Issues in the Intersection Between Software Engineering and Databases*, Sorrento, Italy, 16.–17. May 1994.

[206] U. Kelter and M. Monecke. Eine Architektur zur effizienten Konstruktion verteilter datenbankbasierter Anwendungen. In *Proc. Smalltalk und Java in Industrie und Ausbildung (STJA'99)*, Erfurt, 28.–30.Sept. 1999. Universität Ilmenau.

[207] U. Kelter, M. Monecke, and D. Platz. Realisierung von verteilten Editoren in Java auf Basis eines aktiven Repositories. In *Proc. Java-Informationstage 1998 (JIT'98)*, pages 340–353. Springer, 1998.

[208] U. Kelter, M. Monecke, and D. Platz. Constructing Distributed SDEs using an Active Repository. In Gray et al. [134].

[209] U. Kelter, M. Monecke, and M. Schild. Do we need 'agile' Software Development Tools? In *Proc. Net.ObjectDays 2002*, Oct. 2002.

[210] C. F. Kemerer. How the Learning Curve Affects CASE Tool Adoption. *IEEE Software*, 9(3):23–28, May 1992.

[211] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a Graph Oriented (Software) Engineering Database System. *Information Systems*, 20(1):21–52, 1995.

[212] P. Klein and A. Schürr. Constructing SDEs with the IPSEN Meta Environment. In Ebert and Lewerentz [92], pages 2–10.

[213] H.-U. Kobialka and C. Lewerentz. User Interfaces Supporting the Software Process. In Gruhn [140], pages 60–74.

[214] M. Koskinen. A Metamodelling Approach to Process Concept Customization and Enactability in MetaCASE, 1997. Presented at the ICIS Doctoral Consortium, Atlanta, Georgia, USA, 11.–14. Dec. 1997.

[215] M. Koskinen and P. Marttiin. Developing a Customizable Process Modelling Environment: Lessons Learnt and Future Prospects. In Gruhn [140], pages 13–27.

[216] B. Krishnamurthy and N. S. Barghouti. Provence: A Process Visualisation and Enactment Environment. In I. Sommerville and M. Paul, editors, *Proc. of the* 4^{th} *European Software Engineering Conference*, volume 717 of *Lecture Notes in Computer Science*, pages 451–465. Springer-Verlag, Sept. 1993.

[217] P. Lago and R. Conradi. Transaction planning to support coordination. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, pages 145–151. Lecture Notes in Computer Science 1005, Springer, Oct. 1995.

[218] J.-O. Larsen and P. Lago. Transaction Technology for Process Modelling. In Schäfer [288], pages 214–219.

[219] C. Lassenius, R. Sulonen, K. Alho, and V. Wäyrynen. Integrating Process Modeling with Configuration Management. In *Proc.* 5^{th} *Int. Workshop on Software Configuration Management (SCM5)*, Seattle, Washington, USA, Apr.24.–25. 1995.

[220] A. Laux and L. Martin. XML:DB XUpdate Specification Working Draft. `http://www.xmldb.org/xupdate/xupdate-wd.html`, Sept. 2000.

[221] D. Lee and W. W. Chu. Comparative Analysis of six XML Schema Languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3):76–87, Sept. 2000.

[222] D. Lending and N. L. Chervany. The Use of CASE Tools. In R. Agarwal, editor, *Proceedings of the ACM SIGCPR Conference (SIGCPR-98)*, pages 49–58, New York, Mar. 26–28 1998. ACM Press.

[223] U. Leonhardt, J. Kramer, B. Nuseibeh, and A. Finkelstein. Decentralised Process Enactment in a Multi-Perspective Development Environment. In *Proc. of the* 17^{th} *Int. Conference on Software Engineering*, pages 255–264, Apr. 1995.

[224] B. Lerner. TESS: Automated Support for the Evolution of Persistent Types. In *1997 International Conference on Automated Software Engineering*, pages 172–182. IEEE Computer Society Press, 1997.

[225] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23:111–122, Nov. 1993.

[226] C. Liu and R. Conradi. Process Modelling Paradigms: An Evaluation. In A. Fuggetta, R. Conradi, and V. Ambriola, editors, *Proc. of the 1st European Workshop on Software Process Modeling*, pages 39–52. Associazione Italiane per l'Informatica ed il Calcolo Automatico, 1991.

[227] L. Liu and E. Horowitz. A Formal Model for Software Project Management. *IEEE Transactions on Software Engineering*, 15(10):1280–1293, Oct. 1989.

[228] C. M. Lott. Process and measurement support in SEEs. *Software Engineering Notes*, 18(4):83–93, Oct. 1993.

[229] C. M. Lott. A Controlled Experiment to Evaluate On-Line Process Guidance. *Empirical Software Engineering: An International Journal*, 2(3):269–289, 1997.

[230] C. M. Lott, B. Hoisl, and H. D. Rombach. The use of roles and measurement to enact project plans in MVP-S. In W. Schäfer, editor, *Proc. of the Fourth European Workshop on Software Process Technology*, pages 30–48, Noordwijkerhout, The Netherlands, Apr. 1995. Lecture Notes in Computer Science Nr. 913, Springer–Verlag.

[231] C. M. Lott and H. D. Rombach. Measurement-based Guidance of Software Projects Using Explicit Project Plans. *Information and Software Technology*, 35(6/7):407–419, June/July 1993.

[232] K. Lyytinen, P. Marttiin, J.-P. Tolvanen, M. Jarke, K. Pohl, and K. Weidenhaupt. Bridging the Islands of Automation. In *Proc.* 8^{th} *Annual Workshop on Information Technologies and Systems (WITS'98).*, volume TR-19 of *Computer Science and Information System Reports, Technical Reports*. University of Jyväskylä, 1998.

[233] J. Mack. Was Projektmanager von Expeditionen lernen können.... In *Proc. Net.ObjectDays 2001*, pages 77–86, Sept. 2001.

[234] T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, Mar. 1994.

[235] G. Manson, S. North, and A. Alghamdi. Core Objects Required for a Generic CASE Repository. In *Method Engineering* [40], pages 186–190.

[236] P. Marttiin. Towards Flexible Process Support with a CASE Shell. In G. M. Wijers, S. Brinkkemper, and A. I. Wasserman, editors, *Proc. 6^{th} Int. Conference on Advanced Information Systems Engineering (CAiSE'94)*, volume 811 of *Lecture Notes in Computer Science*, pages 14–27, New York, NY, USA, 1994. Springer-Verlag Inc.

[237] P. Marttiin. Can Process-Centred Environments Provide Customised Process Support Needed in metaCASE? A Literature Review. In *Proc. 1^{st} Int. Workshop on the Many Facets of Process Engineering*, pages 165–180, 1997.

[238] P. Marttiin, F. Harmsen, and M. Rossi. A Functional Framework for Evaluating Method Engineering Environments: The Case of Maestro II/Decamerone and MetaEdit+. In *Method Engineering* [40], pages 63–86.

[239] P. Marttiin and M. Koskinen. Similarities and Differences of Method Engineering and Process Engineering Approaches. In *Effective Utilization and Management of Emerging Information Technologies*, pages 420–424. Idea Group Publishing, 1998.

[240] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The Action Workflow Approach to Workflow Management Technology. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, Emerging technologies for cooperative work, pages 281–288, Toronto, Ontario, 1992. ACM Press.

[241] M. Merz, B. Liberman, and W. Lamersdorf. Using Mobile Agents to Support Interorganizational Workflow Management. *Applied Artificial Intelligence*, 11(6):551–572, 1997.

[242] S. Meyers. Difficulties in Integrating Multiview Development Systems. *IEEE Software*, 8(1):49–57, Jan. 1991.

[243] P. Mi and W. Scacchi. A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):283–294, Sept. 1990.

[244] P. Mi and W. Scacchi. Process Integration in CASE Environments. *IEEE Software*, 9(2):45–53, Mar. 1992.

[245] M. Monecke. *Die Erweiterung der Software-Entwicklungsumgebung ToolFrame um eine Werkzeug-Datenbank*. Diplomarbeit. Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 1996.

[246] M. Monecke. Komponentenbasierte Konstruktion flexibler Software-Entwicklungswerkzeuge. In K. Mehlhorn and G. Snelting, editors, *Proc. Informatik 2000 — Jahrestagung der Gesellschaft für Informatik*, pages 93–107. Springer, Sept. 2000.

[247] M. Monecke. Prozeßmodell-gestützte Adaptierung von CASE-Werkzeugen. In *Proc. GI Informatiktage 2001*, pages 166–169, Bad Schussenried, 2001.

[248] M. Monecke and U. Kelter. Eine integrierte Werkzeugsammlung für den Einsatz in der Softwaretechnik-Ausbildung. Internes Memorandum, Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 2000.

[249] M. Monecke and U. Kelter. Ein repositorybasierter Ansatz für Prozeßunterstützung in CASE-Werkzeugen. In *Proc. Net.ObjectDays 2001*, pages 50–66, Sept. 2001.

[250] C. Montangero, editor. *Proc. 5^{th} European Workshop on Software Process Technology (EWSPT'96)*, volume 1149, New York, NY, USA, Oct. 1996. Springer.

[251] C. Montangero and V. Ambriola. OIKOS: Constructing Process-Centred SDEs. In Nuseibeh et al. [262], pages 131–151.

[252] B. P. Munch, R. Conradi, J.-O. Larsen, N. N. Minh, and P. H. Westby. Integrated Product and Process Management in EPOS. *Journal of Integrated CAE*, 1995.

[253] M. Nagl. Software–Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien. *Informatik Spektrum*, 16(5):273–280, Oct. 1993.

[254] G. Nijssen and T. Halpin. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*. Prentice-Hall, New York, 1989.

[255] H. W. Nissen, M. A. Jeusfeld, M. Jarke, G. V. Zemanek, and H. Huber. Managing Multiple Requirements Perspectives with Metamodels. *IEEE Software*, 13(2):37–48, Mar. 1996.

[256] J. Noak and B. Schienmann. Objektorientierte Vorgehensmodelle im Vergleich. *Informatik Spektrum*, 22(3):166–180, June 1999. in German.

[257] T. Norman, N. Jennings, P. Faratin, and A. Mamdani. Designing and Implementing a Multi-Agent Architecture for Business Process Management. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193 of *LNAI*, pages 261–276. Springer-Verlag: Heidelberg, Germany, Aug.12–13 1997.

[258] B. Nuseibeh. Meta-CASE Support for Method-Based Software Development. In *Proc.* 1^{st} *Int. Congress on Meta-CASE*. University of Sunderland, UK, Jan. 1995.

[259] B. Nuseibeh. To Be and Not to Be: On Managing Inconsistency in Software Development. In *Proc. of the* 8^{th} *Int. Workshop on Software Specification and Design (IWSSD-8)*, pages 164–169, 1996.

[260] B. Nuseibeh and A. Finkelstein. ViewPoints: A Vehicle for Method and Tool Integration. In G. Forte, N. H. Madhavji, and H. A. Müller, editors, *5th Int. Work. Computer-Aided Software Engineering*, pages 50–60, 6–10 July 1992.

[261] B. Nuseibeh, A. Finkelstein, and J. Kramer. Fine-Grain Process Modelling. In J. C. Wileden, editor, *Proc.* 7^{th} *Int. Workshop on Software Specification and Design*, pages 42–46, Redondo Beach, California, Dec. 1993. IEEE Computer Society Press.

[262] B. Nuseibeh, A. Finkelstein, and J. Kramer, editors. *Software Process Modelling and Technology*. John Wiley and Sons, 1994.

[263] B. Nuseibeh, J. Kramer, A. Finkelstein, and U. Leonhardt. Decentralised process modelling. In W. Schäfer, editor, *Proc.* 4^{th} *European Workshop on Software Process Technology*, pages 185–188. Lecture Notes in Computer Science Nr. 772, Springer–Verlag, Apr. 1995.

[264] D. Ohst. Ein Konfigurationskonzept für Änderungsmanagement in verteilten mehrbenutzerfähigen Software-Entwicklungsumgebungen. Vorschlag für eine Dissertation, Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 2001.

[265] OMG. *OMG Unified Modeling Language Specification, Version 1.3*. Object Management Group (OMG), Mar. 2000.

[266] F. Oquendo. SCALE: Process Modelling Formalism and Environment Framework for Goal-directed Cooperative Processes. In *Proc.* 7^{th} *Int. Conference on Software Development Environments (SEE'95)*, Noordwijkerhout, Netherlands, Apr. 1995. IEEE Computer Society Press.

[267] L. J. Osterweil. Software Processes are Software Too. In *Proc.* 9^{th} *Int. Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.

[268] R. Petrasch. Style Guides am Beispiel der Java Look and Feel Design Guidelines von Sun Microsystems - eine kritische Betrachtung. *Softwaretechnik-Trends*, 20(1):17–21, Feb. 2000.

[269] C. Petrie, S. Goldmann, and A. Raquet. Agent-based project management. *Springer LNAI special volume on "Artificial Intelligence Today"*, 1500, 1998.

[270] D. Platz. *Ein Werkzeugtransaktionskonzept für Objekt-Managementsysteme als Basis von Software-Entwicklungsumgebungen*. Shaker Verlag, Aachen, 1999.

[271] D. Platz and U. Kelter. Konsistenzerhaltung von Fensterinhalten in Software-Entwicklungsumgebungen. *Informatik Forschung und Entwicklung*, 12(4):196–205, Dezember 1997.

[272] J. N. Pocock. VSF and its Relationship to Open Systems and Standard Repositories. In A. Endres and H. Weber, editors, *Proc. Software Development Environments and CASE Technology*, volume 509 of *Lecture Notes in Computer Science*, pages 53–68, Berlin, Germany, June 1991. Springer.

[273] K. Pohl and K. Weidenhaupt. A Contextual Approach for Process-Integrated Tools. In M. Jazayeri and H. Schauer, editors, *Proc. 6^{th} European Software Engineering Conference (ESEC/FSE'97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 176–192. Springer / ACM Press, 1997.

[274] K. Pohl, K. Weidenhaupt, R. Dömges, P. Haumer, and M. Jarke. Prozeßintegration in PRIME: Modelle, Architektur und Vorgehensweise. In *Beiträge der GI-Fachtagung Softwaretechnik 1998, Paderborn*, pages 42–52. Gesellschaft für Informatik; Fachausschuß 2.1 (Softwaretechnik und Programmiersprachen) und Softtec NRW e.V. unter Federführung der GI-Fachgruppe 2.1.9 (Objektorientierte Software-Entwicklung), 1998.

[275] K. Pohl, K. Weidenhaupt, R. Dömges, P. Haumer, M. Jarke, and R. Klamma. Process-Integrated (Modelling) Environments (PRIME): Foundation and Implementation Framework. *Transactions on Software Engineering and Methodology*, 1999.

[276] PostgreSQL Web Site. `http://www.de.postgresql.org`.

[277] A. Powell, A. Vickers, E. Williams, and B. Cooke. A Practical Strategy for the Evaluation of Software Tools. In *Method Engineering* [40], pages 165–185.

[278] W. Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks.* dpunkt Verlag für digitale Technologie, Heidelberg, 1997.

[279] *Proc. 7^{th} Int. Conference on Software Development Environments (SEE'95)*, Noordwijkerhout, Netherlands, Apr. 1995. IEEE Computer Society Press.

[280] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66, July 1990.

[281] S. P. Reiss. Simplifying Data Integration: The Design of the Desert Software Development Environment. In *18th International Conference on Software Engineering*, pages 398–407, Berlin - Heidelberg - New York, Mar. 1996. Springer.

[282] S. P. Reiss. Software Tools and Environments. *ACM Computing Surveys*, 28(1):281–284, Mar. 1996.

[283] J. Robie, J. Lapp, and D. Schach. *XML Query Language (XQL).* WWW The Query Language Workshop (QL), Cambridge, MA, Dec. 1998. `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[284] C. Rolland and N. Prakash. A Proposal for Context-Specific Method Engineering. In *Method Engineering* [40], pages 191–208.

[285] P. Rook. Controlling Software Projects. *Software Engineering Journal*, pages 108–117, Jan. 1986.

[286] J. Rosenberg. *Dictionary of Computers, Information Processing, and Telecommunications.* John Wiley & Sons, 2^{nd} edition, 1987.

[287] S. Sachweh and W. Schäfer. Version Management for Tightly Integrated Software Engineering Environments. In *Proc. 7^{th} Int. Conference on Software Engineering Environments.* IEEE Computer Society Press, 1995.

[288] W. Schäfer, editor. *Proc. 4^{th} European Workshop on Software Process Technology (EWSPT '95)*, volume 913, New York, NY, USA, 1995. Springer-Verlag Inc.

[289] M. Schild. *Integration von Projekt-Management-Funktionen in Software-Entwicklungsumgebungen.* Diplomarbeit. Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 1999.

[290] A. Schleicher, B. Westfechtel, and D. Jäger. Modeling Dynamic Software Processes in UML. Aachener Informatik-Berichte 98-11, RWTH Aachen, 1998.

[291] A. Schürr, A. Winter, and A. Zündorf. Spezifikation und Prototyping graphbasierter Systeme. In *Proc. STT '95*, Braunschweig, 1995.

[292] *Proc. Workshop on Standard Exchange Format*, Limerick, Ireland, June 2000.

[293] T. Shepard, S. Sibbald, and C. Wortley. A Visual Software Process Language. *Communications of the ACM*, 35(4):37–44, Apr. 1992.

[294] H. Singh and J. Han. Requirements for Object Management in Software Engineering Environments. Technical Report 96-02, Peninsula School of Computing, Monash University, Melbourne, Australia, Feb. 1996.

[295] I. Sommerville, G. Kotonya, S. Viller, and P. Sawyer. Process Viewpoints. In Schäfer [288], pages 2–8.

[296] X. Song and L. J. Osterweil. Engineering Software Design Processes to Guide Process Execution. *IEEE Transactions on Software Engineering*, 24(9):759–775, Sept. 1998.

[297] G. Stark, R. C. Durst, and C. W. Vowell. Using Metrics in Management Decision Making. *Computer*, 27(9):42–48, Sept. 1994.

[298] S. M. J. Sutton, P. L. Tarr, and L. J. Osterweil. An Analysis of Process Languages. Technical Report UM-CS-1995-078, University of Massachusetts, Amherst, Computer Science, Aug., 1995.

[299] S. M. Sutton, Jr., L. J. Osterweil, and D. Heimbigner. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.

[300] K. D. Swenson. A Visual Language to Describe Collaborative Work. In E. P. Glinert and K. A. Olsen, editors, *Proc. 1993 IEEE Symposium on Visual Languages*, pages 298–305, Bergen, Norway, Aug. 1993. IEEE Computer Society Press.

[301] A. Taivalsaari. Multidimensional Browsing. In Ebert and Lewerentz [92], pages 11–22.

[302] P. Tarr and L. A. Clarke. PLEIADES: An Object Management System for Software Engineering Environments. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 56–70, Dec. 1993.

[303] A. H. M. ter Hofstede and T. F. Verhoef. Meta-CASE: Is the Game worth the Candle? *Information Systems Journal*, 6(1):41–68, 1996.

[304] R. H. Thayer. Software Engineering Project Management: A Top-Down View. *IEEE Transactions on Software Engineering*, pages 15–50, 1987.

[305] I. Thomas and B. A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, 9(2):29–35, Mar. 1992.

[306] P. F. Tiako. Modelling the Federation of Process Sensitive Engineering Environments: Basic Concepts and Perspectives. In Gruhn [140], pages 132–136.

[307] S. Tichelaar, S. Ducasse, and S. Demeyer. FAMIX: Exchange Experiences with CDIF and XMI. In SEF2000 [292].

[308] J.-P. Tolvanen and K. Lyytinen. Flexible Method Adaptation in CASE: The Metamodelling Approach. *Scandinavian Journal of Information Systems (SJIS)*, 5:51–77, 1993.

[309] J.-P. Tolvanen, M. Rossi, and H. Liu. Method Engineering: Current research directions and implications for future research. In *Method Engineering* [40], pages 296–317.

[310] R. Unland. *Objektorientierte Datenbanken: Konzepte und Modelle.* Internat. Thomson Publ., Bonn, 1995.

[311] G. Valetto and G. E. Kaiser. Enveloping Sophisticated Tools into Process-Centered Environments. *Journal of Automated Software Engineering*, 3:309–345, 1996.

[312] T. F. Verhoef and A. H. M. ter Hofstede. Feasibility of Flexible Information Modelling Support. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *Proc. 7^{th} Int. Conference CAiSE'95 on Advanced Information Systems Engineering*, volume 932 of *Lecture Notes in Computer Science*, pages 168–185, Jyväskylä, Finland, June 1995. Springer-Verlag.

[313] M. Verlage. Multi-View Modelling of Software Processes. In Warboys [318], pages 123–126.

[314] I. Vessey, S. L. Jarvenpaa, and N. Tractinsky. Evaluation of Vendor Products: CASE Tools as Methodology Companions. *Communications of the ACM*, 35(4):90–105, Apr. 1992.

[315] I. Vessey and A. P. Sravanapudi. Case Tools as Collaborative Support Technologies. *Communications of the ACM*, 38(1):83–95, Jan. 1995.

[316] B. Wagner, I. Sluijmers, D. Eichelberg, and P. Ackermann. Black-Box Reuse within Frameworks based on Visual Programming. In *Proc. Component Users Conference (CUC'96)*, Munich, 1996.

[317] L. Wakeman and J. Jowett. *PCTE – The Standard for Open Repositories*. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1993.

[318] B. C. Warboys, editor. *Proc. 3^{rd} European Workshop on Software Process Technology*, volume 772 of *Lecture Notes in Computer Science*, Heidelberg, Sept. 1994. Springer–Verlag.

[319] A. Wasserman. Tool Integration in Software Engineering Environments. In F. Long, editor, *Software Engineering Environments, Proc. International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 131–135, Chinon, France, Sept. 1989. Springer.

[320] M. Welle. *Entwurf und Realisierung eines Werkzeugs für die objektorientierte Modellierung zum Einsatz in der Informatik-Ausbildung*. Studienarbeit. Universität-Gesamthochschule Siegen, Fachbereich Elektrotechnik und Informatik, Praktische Informatik, 2000.

[321] E. F. Weller. Using Metrics to Manage Software Projects. *Computer*, 27(9):27–33, Sept. 1994.

[322] B. Westfechtel. Integrated Product and Process Management for Engineering Design Applications. *Integrated Computer-Aided Engineering*, 3(1):20–35, 1996.

[323] B. Westfechtel. *Models and Tools for Managing Development Processes*, volume 1646 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, 1999.

[324] Y. Yang. Coordination for process support is not enought! In Schäfer [288], pages 205–208.

[325] Y. Yang. Issues on Supporting Distributed Software Processes. In Gruhn [140], pages 143–147.

[326] E. Yourdon. *Modern Structured Analysis*. Prentice Hall International, Englewood Cliffs, 1989.

[327] H. Züllighoven. *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- und Materialansatz*. dpunkt-Verlag, Heidelberg, 1998.

Index