Ein repositorybasierter Ansatz für Prozeßunterstützung in CASE-Werkzeugen

Marc Monecke
Universität Siegen
monecke@informatik.uni-siegen.de

Udo Kelter
Universität Siegen
kelter@informatik.uni-siegen.de

Zusammenfassung. Die Entwicklung von Software-Systemen ist eine komplexe und anspruchsvolle Aufgabe, bei der Software-Entwickler durch Analyse- und Entwurfsmethoden unterstützt werden. Kommerzielle CASE-Werkzeuge können nicht an organisationsspezifisch entwickelte oder erweiterte Methoden oder an rollen- und aufgabenspezifische Anforderungen angepaßt werden. Im Papier stellen wir einen Framework-basierten Ansatz zur Konstruktion einer adaptierbaren Werkzeugunterstützung vor. Neben den Konzepten und Notationen kann auch das Vorgehensmodell beschrieben und zur feingranularen Steuerung des Entwicklungsprozesses genutzt werden. Da CASE-Werkzeuge und prozeßspezifische Management-Werkzeuge mit dem gleichen Framework konstruiert werden und ein gemeinsames Repository zur Datenverwaltung nutzen, ist eine enge Integration der CASE-Werkzeuge mit dem Prozeß bei gleichzeitiger Reduzierung des Implementierungsaufwands möglich.

1 Einleitung und Motivation

Die Entwicklung komplexer Software-Systeme ist heute ohne Werkzeugunterstützung nicht mehr denkbar [10]. In den frühen Software-Entwicklungsphasen werden sog. *Upper-CASE-Werkzeuge* eingesetzt, die den Entwickler beim Erzeugen, Bearbeiten, Prüfen, Analysieren und Transformieren von Software-Spezifikationen im Rahmen der Analyse und des Entwurfs unterstützen. Hinzu kommen Werkzeuge für die Planung und Kontrolle der Entwicklungsaktivitäten. Im vorliegenden Papier beschäftigen wir uns mit der Frage, wie diese Werkzeuge effizient konstruiert, an die Anforderungen eines konkreten Entwicklers mit einer bestimmten Aufgabe und Rolle im Software-Entwicklungsprozeß angepaßt und in einer *Software-Entwicklungsumgebung* (SEU) integriert werden können.

Es existieren zahlreiche CASE-Werkzeuge auf dem Markt. Beim Einsatz dieser Werkzeuge innerhalb eines Projekts ist, neben der mangelnden Integration unterschiedlicher Werkzeuge (bzgl. Benutzungsschnittstellen, Daten und Kon-

trolle [31]), die mangelnde *Adaptierbarkeit* der Werkzeuge an die konkreten Projekterfordernisse ein Problem [22]:

1. Es können nur die vom Werkzeug angebotenen Dokumenttypen, Darstellungen, Kommandos und Konsistenztests verwendet werden; eine Erweiterung ist nicht möglich. Für firmenspezifisch angepaßte oder vollständig neu entwickelte Methoden steht also keine Unterstützung zur Verfügung.
2. Zum Teil sind Vorgehensmodelle im Werkzeug 'hart verdrahtet', die tatsächlichen Arbeitsabläufe im Projekt müssen also an die Vorgaben des Werkzeugs angepaßt werden, auch wenn diese nicht zum Projekt passen oder von den traditionellen Arbeitsabläufen innerhalb der Organisation abweichen. Dadurch erhöht sich auch der Zeit- und Kostenaufwand für die Werkzeugeinführung. Andererseits kann die Verwendung von Werkzeugfunktionen wie Analyse- und Prüfkommandos zu einem bestimmten Zeitpunkt im Prozeß nicht vorgeschrieben werden. Gerade durch solche Funktionen kann die Qualität der bearbeiteten Dokumente erhöht werden, sie werden aber nur selten tatsächlich genutzt [20]. Daher wäre es z.B. sinnvoll sicherzustellen, daß vor dem Abschluß einer Entwurfsaufgabe eine Konsistenzprüfung durchgeführt wird, abgeleitete Dokumente wie Quelltexte oder Dokumentationen aber erst generiert werden können, wenn diese Prüfung erfolgreich war.
3. Die Optimierung des Funktionsumfangs eines Werkzeugs für eine bestimmte Aufgabe/Rolle ist meist nicht möglich. Ein Entwickler benötigt für eine konkrete Aufgabe meist nur einen Bruchteil der Werkzeugfunktionalität, wird aber stets mit dem vollen Funktionsumfang konfrontiert. Hierdurch wird die Handhabung des Werkzeugs erschwert, die Bereitschaft des Entwicklers zur und seine Effizienz bei der Nutzung des Werkzeugs werden reduziert [5]. Dies trifft insbesondere auf Benutzer zu, die unerfahren in der Anwendung von Werkzeug und Methode sind oder diese nur gelegentlich einsetzen. Gerade in der Ausbildung werden Anfänger durch kommerzielle Werkzeuge meist überfordert [25]. Gutes Beispiel hierfür sind Werkzeuge für die *Unified Modeling Language* (UML) [28] mit ihren zahlreichen Dokumenttypen und der großen Mächtigkeit der Sprache.

Eine Lösung der genannten Probleme bieten zum einen *Meta-CASE-Systeme* [15], mit denen Werkzeuge, die die Konzepte und Notationen einer gegebenen Methode unterstützen, effizient konstruiert werden können. Zum anderen erlauben *Prozeßorientierte Software-Entwicklungsumgebungen* (*Process-centered Software Engineering Environment*, PSEU) [2] die explizite Beschreibung des Entwicklungsprozesses in Form eines *Prozeßmodells*. Das Prozeßmodell wird von einer *Prozeßmaschine* interpretiert, die den Verlauf des Entwicklungsprozesses überwacht und steuert. Werden CASE-Werkzeuge in eine PSEU integriert, handelt es sich um *prozeßintegrierte* Werkzeuge [6], die einerseits durch die PSEU abhängig vom konkreten Prozeßzustand beeinflußt werden (z.B. bzgl. der angebotenen Kommandos und Darstellungen) und andererseits der PSEU Rückmeldungen über die vom Benutzer durchgeführten Operationen liefern.

Die nachträgliche Prozeßintegration vorhandener Werkzeuge ist nur sehr eingeschränkt möglich, da die Werkzeuge zum einen keine geeigneten Schnittstellen anbieten, andererseits die Interaktion zwischen Werkzeug und PSEU (sofern möglich) sehr aufwendig und fehleranfällig implementiert werden müßte. Die Prozeßintegration muß also bereits beim Entwurf der Werkzeuge berücksichtigt werden.

Wir nutzen den Werkzeug-Konstruktionsansatz *genform* [24] zum Bau von PSEU. Mit *genform* werden sowohl prozeßintegrierte CASE-Werkzeuge, Planungs- und Managementwerkzeuge, als auch die Prozeßmaschine konstruiert, was die Integration der Werkzeuge untereinander und mit dem Prozeß erleichtert und den Implementierungsaufwand stark reduziert. Gleichzeitig ist eine flexible Erweiterung und Anpassung der Werkzeuge an die jeweiligen Besonderheiten des Prozesses möglich. Die Anpassung geschieht zum großen Teil automatisch, da die Werkzeugspezifikationen direkt aus dem Prozeßmodell generiert werden. Zentraler Bestandteil der mit *genform* konstruierten PSEU ist das Repository, das neben den Softwaredokumenten auch den Prozeßzustand verwaltet und somit zur (Prozeß-)Integration der CASE-Werkzeuge dient. Die Dienste des Repositories werden für die Realisierung der Funktionalität von Werkzeugen und Prozeßmaschine direkt genutzt.

Gliederung. Auf verwandte Arbeiten im Bereich der Methodenunterstützung wird in Abschnitt 2 kurz eingegangen. Abschnitt 3 erläutert die Werkzeugkonstruktion mit *genform*. In Abschnitt 4 werden Erweiterungen vorgestellt, mit denen die feingranulare Prozeßunterstützung realisiert wird. In Abschnitt 5 gehen wir kurz auf die Implementierung ein; Abschnitt 6 faßt das Papier zusammen.

2 Andere Ansätze

Meta-CASE-Systeme. Es existieren zahlreiche Meta-CASE-Systeme, zum Teil als Forschungsprototypen, zum Teil im kommerziellen Einsatz. Wesentlicher Bestandteil eines Meta-CASE-Systems ist ein *generischer Editor*, mit dem Dokumente verschiedenen Typs bearbeitet werden können [1]. Um eine konkrete 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).

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 [7]. In diesem Fall steigt der Aufwand für die Adaptierung des Werkzeugs überproportional, da die Einschränkungen des Meta-Systems (falls überhaupt möglich) umgangen werden müssen.

Frameworks zur Konstruktion von CASE-Werkzeugen wie *MetaMOOSE* [7] und *Serendipity* [9] haben dieses Problem nicht, da die objektorientierten Programmiersprachen Wiederverwendung und Abstraktion direkt unterstützen.

Eine wichtige Frage, mit der sich uns bekannte Ansätze nicht beschäftigen,

ist, wie leicht *Varianten* eines Werkzeugs erzeugt werden können, die jeweils auf eine bestimmte Benutzungssituation im Software-Entwicklungsprozeß ausgelegt sind. Beispiel:

- Beim Systementwurf wird ein graphischer Editor für Klassendiagramme benötigt, um ein objektorientiertes Modell zu bearbeiten. Im Editor sollten auch Kommandos zum Durchführen von Konsistenzprüfungen und zum Generieren von Quelltexten und Dokumentationen verfügbar sein.
- Zur Qualitätssicherung müssen zusätzlich Messungen durchgeführt und die Ergebnisse visualisiert werden können, etwa durch das Markieren fehlerhafter Klassen und kritischer Bereiche im Entwurf.
- Einen Manager interessieren die Aufwände und Kosten, die bei der Bearbeitung von Klassen oder Paketen entstanden, sowie der Fertigstellungsgrad der einzelnen Komponenten in Relation zum geplanten Fertigstellungstermin. Außerdem möchte er feststellen, welcher Entwickler eine Komponenten bearbeitet. Hier könnten also unterschiedliche Darstellungen für Klassen verwendet und zusätzliche Eigenschaften in separaten Dialogen angezeigt werden.
- Im Rahmen der Versionsverwaltung müssen Konfigurationen erzeugt und Unterschiede zwischen verschiedenen Versionen eines Klassendiagramms visualisiert und ggf. abgeglichen werden können.

Alle Werkzeuge für die genannten Aufgaben arbeiten auf dem objektorientierten Modell. Sie unterscheiden sich aber in ihren Datenmodellen (da sie zusätzliche Informationen z.B. über Qualität und Kosten benötigen), bieten zusätzliche oder erweiterte Darstellungen (z.B. Anzeige von Fehlern) mit den zugehörigen Interaktionsmöglichkeiten und Kommandos an (z.B. Auslösen von Prüfungen, Versionieren eines Dokuments).

Es ist hier nicht sinnvoll, die verschiedenen Werkzeuge unabhängig voneinander (und zum großen Teil redundant) zu spezifizieren. Bei der Nutzung eines Frameworks würden komplexe Vererbungshierarchien entstehen, in denen die Auswirkungen von Änderungen schwer nachvollziehbar sind. Der Meta-CASE-Ansatz sollte es daher ermöglichen, die unterschiedlichen *Werkzeugeigenschaften* zu spezifizieren und einerseits die Gemeinsamkeiten in den verschiedenen Varianten wiederzuverwenden, aber andererseits auch umfangreiche Erweiterungen der Werkzeuge zulassen.

Prozeßorientierte Umgebungen. Prozeßorientierte Umgebungen (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.

Eine große Zahl von PSEU ist entwickelt und beschrieben worden (z.B. in [2, 27]). 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.

In einer PSEU stehen Werkzeuge zur Verfügung, mit denen Entwickler ihre Aufgabenlisten betrachten, Aufgaben ausführen und Rückmeldungen an den Prozeß schicken. Diese Werkzeuge sollten möglichst nahtlos in die Produktionsumgebung des Entwicklers eingebunden sein. Manager analysieren und überwachen mit Management-Werkzeugen 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. Auch die von den Entwicklern genutzten CASE-Werkzeuge sollten mit der PSEU integriert sein, so daß z.B. Aktionen wie das Erzeugen von Dokumenten oder die Änderung von Dokumentzuständen der Prozeßsteuerung nicht manuell mitgeteilt werden müssen.

Fazit. Nur wenige Arbeiten versuchen, eine Adaptierung der Werkzeuge an Konzepte, Notationen *und* Prozeßmodelle zu ermöglichen, so daß eine durchgängige Methodenunterstützung effizient konstruierbar ist [23]. Hierzu gehören [19] und [22], in denen das Meta-CASE-System MetaEdit+ nachträglich um eine Prozeßunterstützung erweitert wurde, und [26], wo Sichten (*viewpoints*) für die verschiedenen Rollen im Prozeß definiert werden können, die jeweils Produkte, Werkzeuge und Aufgaben eines Prozeßteilnehmers kapseln und somit sehr *feingranulare* Prozeßmodelle [21] enthalten.

Im nächsten Abschnitt beschreiben wir den Werkzeug-Konstruktionsansatz *genform*, der für die effiziente Konstruktion von CASE-Werkzeugen mit durchgängiger Methodenunterstützung genutzt wird.

3 Dokumente und Werkzeuge

Beim Bau von CASE-Werkzeugen für eine gegebene Methode müssen drei Fragen geklärt werden:

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

3.1 Werkzeugschemata

In *genform* werden das Datenmodell der Dokumente und die Eigenschaften der zugehörigen Werkzeuge in *Werkzeugschemata* (WZS) definiert [24].

Ein WZS enthält

- das *Datenmodell* der Dokumente, die das Werkzeug bearbeitet. Das Datenmodell besteht aus Objekt- und Beziehungstypen, 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. Werkzeuge können zur Laufzeit auf diese Parameter zugreifen und die Werte interpretieren. Durch Werkzeugparameter kann z.B. das Aussehen von Formularen und die graphische Darstellung von Dokumenten beeinflußt und die Zusammensetzung der Werkzeuge aus Werkzeugkomponenten gesteuert werden.
- *Benutztbeziehungen* zu anderen WZS, mit denen einzelne Werkzeuge zu einer Umgebung verknüpft werden. Dem Benutzer präsentieren sich solche Beziehungen z.B. als Menüeinträge zum Starten eines anderen Werkzeugs.

Durch *Import-Beziehungen* zwischen WZS können Typdefinitionen in andere WZS übernommen und dort ggf. 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.

3.2 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 (ähnlich einem Syntaxbaum) rekursiv durch Objekt- und Beziehungstypen beschrieben. Die Attribute von Objekt- und Beziehungstypen enthalten die Eigenschaften von Dokumenteinträgen und Beziehungen, wie Name, Beschreibung oder Typ. In Abb. 1 ist ein vereinfachtes Werkzeugschema für einen graphischen Klassendiagramm-Editor skizziert.

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

Mit *genform* konstruierte Umgebungen verwenden ein strukturell objektorientiertes Datenbank-Managementsystem als Repository. Datenbankschemata (sog. *Schema Definition Sets*, SDS) werden in einer textuellen Datendefinitionssprache beschrieben. Eine solche SDS-Beschreibung kann direkt aus dem Modell aus Abb. 1 generiert werden. Durch eine Folge von SDS wird die aktuelle Sicht (das

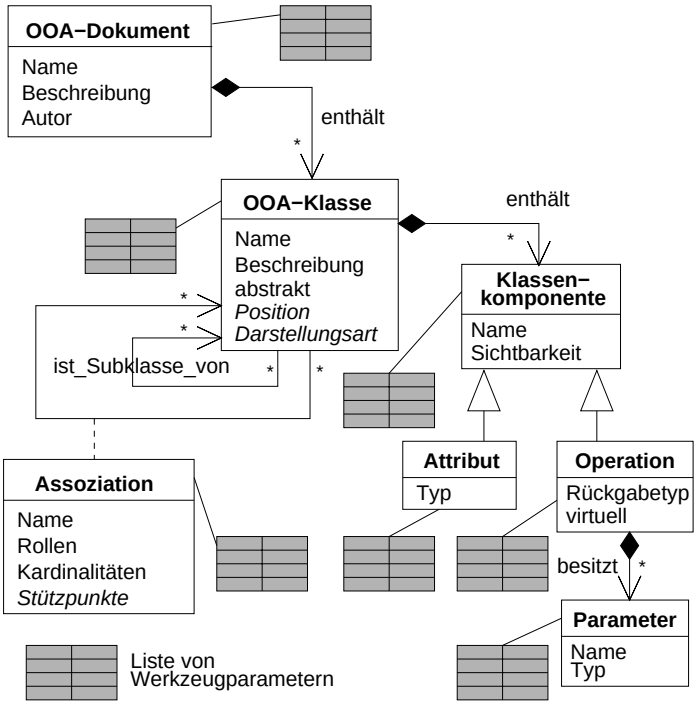


Abbildung 1. Vereinfachtes Werkzeugschema für Klassendiagramm-Editor

Arbeitsschema) eines Werkzeugs definiert. Sie enthält die Vereinigung aller Typen aus den angegebenen SDS.

Zu den WZS für den Klassenbrowser und den graphischen Editor gibt es also korrespondierende SDS, in denen das Datenmodell der Dokumente beschrieben ist. Den Typen im SDS sind die Werkzeugparameter zugeordnet. Die Schemaverwaltung des Repositories wird damit direkt zur Verwaltung der WZS genutzt, muß also nicht in *genform* implementiert werden.

3.3 Werkzeugkomponenten

Die Werkzeugfunktionalität ist in den *Werkzeugkomponenten* des *genform*-Frameworks implementiert. Die Struktur des Frameworks entspricht i.w. der bei GUI-Frameworks üblichen: Es gibt Komponenten für Fensterrahmen, Fensterinhalte, wie Formulare und graphische Zeichenflächen, GUI-Elemente wie Schaltknöpfe und Eingabefelder. Die Komponenten beschränken (und spezialisieren) sich dabei auf die in CASE-Werkzeugen benötigte Funktionalität. Die *genform*-Komponenten haben zwei Besonderheiten:

1. Sie sind *generisch*, d.h. sie interpretieren Datenmodell [4] und Werkzeugparameter aus dem WZS. Eine Komponente muß also nicht auf Programmiersprachen-Ebene adaptiert werden, sondern paßt ihr Verhalten

an die Informationen aus dem WZS an.

Beispiele: Ein generisches Formular kann die Attribute eines Objekts aus dem Datenmodell ermitteln und für jedes Attribut ein passendes Eingabefeld (Textfeld, *check box* o.ä.) erzeugen. Die Darstellung des Formulars (Text und Plazierung von *labels*, Anzahl Zeilen von Texteingabefeldern) wird über Werkzeugparameter spezifiziert, die den Objekt- bzw. Attributtypen zugeordnet sind.

Ein graphischer Editor ermittelt die Typen von Dokumenteinträgen und Beziehungen aus dem Datenmodell und erzeugt die zugehörigen Kommandos zum Einfügen von Einträgen und Beziehungen. Die Werkzeugkomponenten, die Dokumenteinträge graphisch darstellen (z.B. Klassensymbol, Vererbungspfeil), werden über Werkzeugparameter spezifiziert und zur Werkzeuglaufzeit geladen und instantiiert.

2. Sie nutzen die Dienste des Repositories zur Verwaltung und Manipulation der Dokumente direkt aus: Änderungen, die der Benutzer mit einem Werkzeug macht, werden *direkt* im Repository durchgeführt. Der *Benachrichtigungsmechanismus* des Repositories sorgt dafür, daß Zustand und Anzeige der Werkzeuge nach Änderungen im Repository automatisch aktualisiert werden. Die Werkzeuge müssen sich also nicht um Änderungspropagation kümmern (für weitere Details der Architektur sei auf [18] verwiesen). Durch die gewählte Architektur können die Rechteverwaltung, Sperr- und Konsistenzsicherungsmechanismen des Repositories direkt in den Werkzeugen genutzt werden, so daß sich der Implementierungsaufwand für Framework und Werkzeuge deutlich verringert. Durch die angebotenen Dienste sind die Werkzeuge mehrbenutzerfähig und verteilt einsetzbar.

Mit Werkzeugschemata können sehr schnell Werkzeuge für einen gegebenen Dokumenttyp spezifiziert werden[1].

Das in Abschnitt 2 beschriebene Problem der nicht vorhergesehenen Erweiterungen, das zu einem starken Anstieg des Implementierungsaufwands für die Werkzeuge führt, wird durch die Möglichkeit gelöst, Framework-Komponenten zu erweitern. Die Erweiterungen werden auf hohem Abstraktionsniveau vorgenommen, da die Konzepte des Frameworks genutzt werden können, wobei die Lernkurve des Werkzeugentwicklers mit dem Umfang der nötigen Erweiterung linear ansteigt: Im häufigsten Fall werden nur 'Blätter' in der Klassenhierarchie mit wenigen Beziehungen zu anderen Komponenten erweitert (z.B. Komponenten für die graphische Darstellung von Dokumenteinträgen oder dokumentspezifische Kommandos); hierzu ist nur die Kenntnis weniger Framework-Klassen nötig. Bei der Erweiterung 'innerer' Klassen (wie der Zeichenfläche des graphischen Editors) ist eine umfangreichere Kenntnis des Frameworks nötig.

Durch die Erweiterung von Framework-Komponenten entstehen *werkzeugspezifische Komponenten*. Diese werden über Werkzeugparameter in Werkzeuge

[1] Hierfür stehen prototypische Meta-Werkzeuge zur Verfügung, die ebenfalls mit *genform* konstruiert wurden und mit denen der Werkzeugentwickler Werkzeugschemata entwerfen kann. Auf diese Werkzeuge wird hier nicht weiter eingegangen.

eingebunden, z.B., indem Dokumenteinträgen werkzeugspezifische Darstellungskomponenten zugewiesen werden. Um die Komponenten zu nutzen, müssen also keine weiteren Klassen modifiziert werden.

4 Prozeßunterstützung mit *genform*

Im vorangegangenen Abschnitt haben wir beschrieben, wie mit *genform* CASE-Werkzeuge konstruiert werden können, die dokumentspezifische Darstellungen und Kommandos enthalten, und damit die in einer Methode definierten Konzepte und Notationen unterstützen.

Zur vollständigen Unterstützung einer Methode muß auch ihr Vorgehens- oder Prozeßmodell berücksichtigt werden. Wird dieses hinreichend formalisiert, kann es von einer Prozeßmaschine ausgeführt werden. Die Prozeßmaschine verwaltet dann einen virtuellen 'internen' Prozeßzustand, der durch Rückmeldungen von Entwicklern und Managern mit dem realen 'externen' Prozeßzustand abgeglichen wird, und leitet Entwickler und Manager dabei an, ihre Aufgaben gemäß dem Prozeßmodell durchzuführen (*on-line guidance*). Ein wichtiger Aspekt ist hier, daß die CASE-Werkzeuge mit Prozeßmodell und Prozeßzustand integriert sein müssen, Entwickler also direkt über ihre Werkzeuge mit dem Prozeß interagieren können.

Im folgenden beschreiben wir, wie *genform* erweitert wurde, um die Modellierung und Ausführung von Entwicklungsprozessen mit prozeßintegrierten Werkzeugen zu unterstützen.

4.1 Prozeßmodellierung

Prozeßmodelle werden in *genform* graphisch dargestellt. Abb. 2 zeigt einen Ausschnitt aus dem Prozeßmodell zur objektorientierten Analyse (OOA) [3].

Elemente des Prozeßmodells. Das Prozeßmodell enthält die *Aufgabentypen* 'finde Vererbungsbeziehung' und 'prüfe Vererbungsbeziehung'. Die Aufgabentypen legen die Eigenschaften konkreter Aufgaben (der Instanzen des Aufgabentyps) fest. Aufgabentypen können durch Unteraufgaben hierarchisch verfeinert werden.

Nachfolgerbeziehungen zwischen Aufgabentypen beschreiben den Kontrollfluß innerhalb des Prozesses. Sie werden durch gefüllte Dreiecke symbolisiert. An Aufgabentypen und Beziehungen werden *Kardinalitäten* notiert, mit denen die erlaubte Zahl der Instanzen vorgegeben wird (in eckigen Klammern).

Jedem Aufgabentyp wird eine *Rolle* zugeordnet (z.B. OODesigner, Programmierer). Die Rolle wird im Symbol des Aufgabentyps angezeigt. Ein Benutzer muß diese Rolle besitzen, um die Aufgabe ausführen zu können.

Die Dokumente, die in einer Aufgabe bearbeitet werden, kann man als Parameter der Aufgabe betrachten. Die 'Signatur' eines Aufgabentyps wird durch die Zuordnung von (Typen von) *Dokumentordnern* spezifiziert. Technisch ist ein Dokumentordner ein Verzeichnis, das Verweise auf Dokumente (bzw. Dokumentteile) eines Typs enthält, auf die im Rahmen der Aufgabe zugegriffen wird.

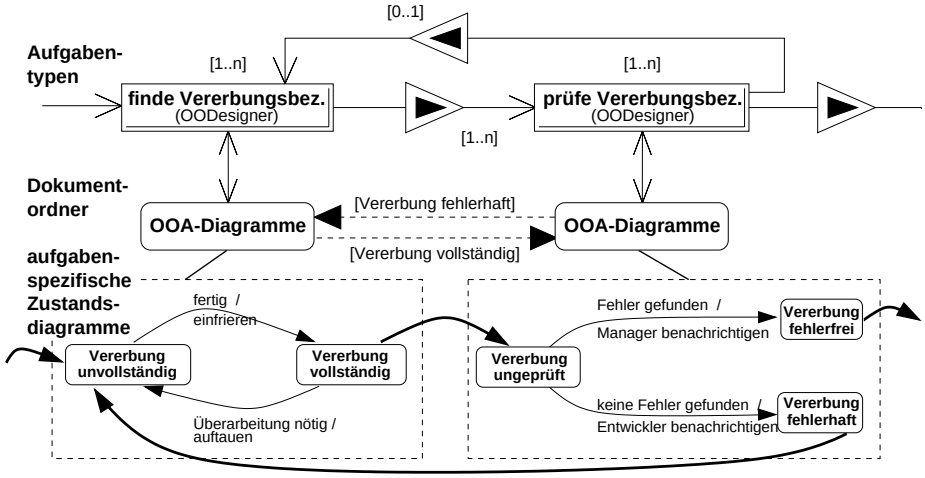


Abbildung 2. Prozeßmodellausschnitt für OOA in *genform*-Notation, lokale Zustandsdiagramme

Analog zu einem Parameter einer Methode kann auch bei Dokumentordnern die Übergabeart spezifiziert werden. Sie legt fest, ob Dokumente von der Aufgabe nur gelesen, nur geschrieben (erzeugt, verändert) oder beides werden. Die Übergabeart wird durch die Pfeilspitzen an der Beziehung zwischen Aufgabentyp und Dokumentordner symbolisiert und bestimmt die Zugriffsrechte, die der Bearbeiter einer Aufgabe an den enthaltenen Dokumenten besitzt.

Transitionen zwischen Ordnern werden durch gestrichelte Pfeile dargestellt. Über eine Transition wird ein Dokument von einer Aufgabe an eine Nachfolgeaufgabe weitergeleitet (ähnlich einer Marke in einem PrT-Netz). Die Weiterleitung von Dokumenten kann über verschiedene Eigenschaften der Transition gesteuert werden; dazu gehört der Zustand, in dem sich die Dokumente befinden müssen (in eckigen Klammern angegeben).

Aufgabentypen und Werkzeuge. Um den Prozeß vollständig zu spezifizieren, muß angegeben werden, welche Werkzeuge für eine Aufgabe benutzt werden sollen. Hierzu werden den Dokumentordnern Werkzeugschemata zugeordnet (in der graphischen Darstellung des Prozeßmodells nicht sichtbar). Da ein Werkzeugschema die Eigenschaften eines Werkzeugs feingranular beschreibt, kann genau festgelegt werden, welche Sicht der Bearbeiter einer Aufgabe auf die Dokumente besitzt und welche Darstellungen und Kommandos das Werkzeug anbietet[2].

[2]Die Menge von Dokumenten und Werkzeugen, die einer Aufgabe zugeordnet sind, wird auch als *Arbeitskontext* (*work context*) bezeichnet, s. z.B. [33]. Natürlich können WZS in verschiedenen Prozeßmodellen wiederverwendet oder prozeßspezifische Varianten entworfen werden.

Zustände. Im Prozeßmodell müssen auch die verschiedenen Zustände des Entwicklungsprozesses beschrieben werden, die zwischen dem initialen Zustand ('Prozeß gestartet') und dem Endzustand ('Prozeß beendet') liegen. Der Zustand des Entwicklungsprozesses setzt sich zusammen aus den Zuständen von Aufgaben und der bearbeiteten Dokumente, die jeweils durch Zustandsdiagramme beschrieben werden.

Genauer sind die Zustandsdiagramme der Dokumente den Dokumentordnern zugeordnet. Ein Dokument (im Beispiel ein OOA-Dokument) hat also in verschiedenen Ordnern unterschiedliche Zustände. Übergänge zwischen den lokalen Zustandsdiagrammen sind in der Abbildung durch einen dicken Pfeil symbolisiert. Sie werden von der Prozeßmaschine ausgelöst, wenn ein Dokument entlang einer Transition von einem Ordner in einen anderen wechselt.

An den Zustandsübergängen sind Bedingungen und Aktionen notiert, durch die festgelegt wird, wann ein Zustandswechsel erfolgen darf bzw. was dabei passiert. Mit den Bedingungen an Zustandsübergängen können auch *feingranulare* Abhängigkeiten zwischen Aufgaben realisiert werden. Beispiel: Eine Bedingung am Übergang von ready nach active im Zustandsdiagramm einer Aufgabe A_2 überwacht die Vorgänger-Aufgabe A_1. Die Bedingung sei erfüllt, sobald A_1 ein Ausgabedokument produziert hat. In diesem Fall wird A_2 aktiviert.

Komponenten. Wie bei der Konstruktion von Werkzeugen werden auch Prozeßmodelle und Mechanismen für ihre Ausführung auf zwei Ebenen konstruiert: Auf der Spezifikationsebene werden Prozesse mit der graphischen Modellierungssprache beschrieben. Das Prozeßmodell kann von der Prozeßmaschine interpretiert und somit eine automatisierte Prozeßunterstützung erreicht werden.

Um eine höhere Flexibilität bei der Adaptierung von Werkzeugen an ein gegebenes Prozeßmodell zu ermöglichen, werden auf der Implementierungsebene Komponenten der Prozeßmaschine (durch Erweiterung vorhandener Komponenten) implementiert. Auf die Architektur der Prozeßmaschine und ihre Erweiterungsmöglichkeiten gehen wir näher im Abschnitt 4.3 ein.

4.2 Instantiierung des Prozeßmodells

Nach Abschluß der Modellierung kann das Prozeßmodell instantiiert und mit der Ausführung[3] des Prozesses begonnen werden. Die Instantiierung umfaßt

- die Einrichtung der 'Laufzeitumgebung' des Prozesses, also das Initialisieren des Repositories, in dem alle Projektdaten zentral verwaltet werden.
- das Bereitstellen der ausführbaren Werkzeuge und das Einrichten der Betriebssystem-Umgebung für deren Einsatz; damit beschäftigen wir uns an dieser Stelle nicht.

[3] Im Englischen wird zwischen *process execution* und *process enactment* unterschieden. Ersteres bedeutet die automatisierte Ausführung des Prozesses, während *enactment* die beteiligten Personen einbezieht, die mit dem Prozeß interagieren, Entscheidungen treffen und den Ablauf beeinflussen. In diesem Sinne meinen wir mit 'Ausführung' des Prozesses *process enactment*.

Dokumente. Im Prozeßmodell sind die Datenmodelle der Dokumente als Teil der Werkzeugschemata enthalten. Damit die Dokumente im Repository verwaltet werden können, müssen passende Datenbankschemata (SDS) erzeugt werden. In unserem Ansatz werden zunächst Textdateien generiert, die die Beschreibung des Schemas in der Datendefinitionssprache des Repositories enthalten. Mit einem Administrationswerkzeug wird dann das Schema im Repository erzeugt.

Wichtig ist dabei, daß die Importbeziehungen zwischen WZS auf Importbeziehungen zwischen den Datenbankschemata abgebildet werden, also keine Redundanzen zwischen den Datenmodellen verschiedener Werkzeugvarianten entstehen.

Management. Die Durchführung eines Projekts läßt sich zerlegen in die Durchführung einzelner Aufgaben. Die Eigenschaften der Aufgabentypen (Rollen, Dokumenttypen, Werkzeuge), ihre Struktur (Unteraufgaben) und Reihenfolge (Nachfolgerbeziehungen) werden durch das Prozeßmodell vorgegeben. Ein laufendes Projekt kann man sich also als dynamisch wachsendes Netz von Aufgaben vorstellen [11].

Betrachtet man das Aufgabennetz als Dokument, so stellt das Prozeßmodell das Datenmodell dieses Dokuments dar. Das Planungswerkzeug des Managers ist somit ein Editor, mit dem man Aufgabennetz-Dokumente bearbeiten kann, der also die verschiedenen Konzepte (Aufgabentypen, Beziehungen zwischen Aufgaben, Dokumentordner) unterstützt und geeignete Darstellungen und Kommandos anbietet.

Mit *genform* kann man die Planungs- und Managementwerkzeuge leicht konstruieren und flexibel z.B. um Kommandos für die Analyse des Prozeßzustands oder zur Reportgenerierung erweitern. Hierzu ist keine Erweiterung des Konstruktionsansatzes oder der Werkzeugarchitektur nötig. Benötigt werden ein Werkzeugschema und ggf. werkzeugspezifische Komponenten. Ersteres wird aus dem Prozeßmodell generiert, wobei Aufgabentypen auf Eintragstypen und Beziehungen zwischen Aufgabentypen auf Beziehungstypen im Dokument abgebildet werden. Durch den gemeinsamen Konstruktionsansatz ist die Integration von Management- und CASE-Werkzeugen einfach möglich. Auch wird der Implementierungsaufwand für das Framework reduziert, da gleiche Framework-Komponenten in den verschiedenen Werkzeugen zum Einsatz kommen.

Benutzer und Rollen. Die im Prozeßmodell definierten Rollen werden auf die hierarchischen Benutzergruppen des Repositories abgebildet. Ein Benutzer kann Mitglied in beliebig vielen Gruppen sein, besitzt aber stets genau eine aktuelle Gruppe aus dieser Menge, die seiner aktuellen Rolle entspricht. Die Gruppenstruktur wird bei der Initialisierung des Repositories anhand der Rollendefinitionen im Prozeßmodell angelegt. Einzelne Benutzer werden mit einem Administrationswerkzeug eingetragen und den verschiedenen Gruppen zugeordnet. Bei der Zuordnung eines Benutzers zu einer Aufgabe wird geprüft, ob er die geforderte Rolle übernehmen kann, also Mitglied in der zugehörigen Gruppe ist. Die Prozeßmaschine setzt die Zugriffsrechte an den Objekten im Repository

und legt somit fest, welche Operationen ein Benutzer oder eine Rolle ausführen kann. Die Verwaltung von Benutzern, Benutzergruppen und Rechten wird dabei vollständig vom Repository übernommen.

Die Werkzeugkomponenten in *genform* prüfen die Rechtesituation an Objekten und deaktivieren ggf. Kommandos (z.B. zum Erzeugen und Löschen von Einträgen, wenn an einem Dokument nur Leserechte bestehen) oder blenden Einträge aus, auf die nicht zugegriffen werden kann.

Beeinflussung des Prozeßzustands. Die Liste der Aufgaben eines Entwicklers wird in seinem *Agenda-Werkzeug* angezeigt, das in seine SEU integriert ist. Hier kann er die Bearbeitung einer Aufgabe beginnen, unterbrechen, wiederaufnehmen und beenden (kurz: den Zustand der Aufgabe ändern) und somit den Prozeßzustand direkt beeinflussen. Über die Aufgabe kann er direkt auf zugeordnete Werkzeuge und Dokumente zugreifen (Arbeitskontext). Indirekt wird der Prozeßzustand durch die Änderung der Zustände von Dokumenten beeinflußt. Eine Prüfaufgabe ist z.B. erledigt, sobald alle zugeordneten Diagramme geprüft wurden. Der Zustand der Aufgabe wird also vom Zustand der Dokumente abgeleitet. Für die Zustandspropagation ist die Prozeßmaschine (Abschnitt 4.3) zuständig.

Durch die Integration der Entwurfswerkzeuge mit dem Prozeß kann der Zustand von Dokumenten direkt in den Entwurfswerkzeugen manipuliert werden, wobei die in den zugehörigen Zustandsdiagrammen definierten Bedingungen berücksichtigt und die festgelegten Aktionen ausgeführt werden.

4.3 Prozeßmaschine

Die Prozeßmaschine interpretiert das Prozeßmodell und kommuniziert mit den CASE- und Managementwerkzeugen über das Repository. Das Verhalten der Prozeßmaschine (*process engine*, PE) wird also im Prozeßmodell festgelegt und kann während der Prozeßausführung nur indirekt (durch erlaubte Änderungen von Aufgaben- und Dokumentzuständen) beeinflußt werden.

Werden die Zustände von Dokumenten und Aufgaben geändert, reagiert die PE hierauf, indem sie ihrerseits Zustände modifiziert (z.B. die Folgeaufgabe einer abgeschlossenen Aufgabe aktiviert), oder Aktionen ausführt. Diese Aktionen können darin bestehen, ein Dokument über eine Transition an die Folgeaufgabe weiterzuleiten, einen Projektteilnehmer über eine Änderung zu informieren, eine Qualitätsmessung durchzuführen, oder mit einem Transformator ein abgeleitetes Dokument zu erzeugen. Die Entwickler und Manager im Prozeß werden also von diesen Routinetätigkeiten entlastet; gleichzeitig wird sichergestellt, daß keine (insbesondere aus Sicht der Qualitätssicherung) erforderliche Aktion vergessen wird.

Im folgenden wird erläutert, wie die PE so konstruiert werden kann, daß ihr Verhalten flexibel an das jeweilige Prozeßmodell anpaßbar ist.

Bedingungen und Aktionen. Den Zustandsübergängen in den Zustandsdiagrammen von Aufgaben und Dokumenten werden Bedingungen und Aktionen

zugeordnet. Durch *Bedingungen* kann z.B. der Kontrollfluß zwischen Aufgaben beeinflußt werden, indem der Nachfolger startet, sobald der Vorgänger ein Ausgabedokument produziert hat. Der Nachfolger muß also nicht bis zum Ende des Vorgängers warten. Bedingungen sind aktive Komponenten, die den Benachrichtigungsmechanismus des Repositories nutzen, um Änderungen zu überwachen, diese auszuwerten, ggf. ihren eigenen Zustand zu ändern und die PE hierüber zu informieren.

Aktionen werden im Rahmen eines Zustandswechsels ausgeführt. Ändert sich der Zustand einer Aufgabe oder eines Dokuments, kann z.B. das Dokument versioniert oder eine Nachricht an den Manager geschickt werden (vgl. Abb. 2). Aktionen können auch Werkzeugkommandos ausführen. Auch Aktionen basieren auf Framework-Komponenten, die ggf. prozeßspezifisch erweitert werden.

Prozeßagenten. Vergleicht man die Beschreibung eines Software-Prozesses mit der eines Informationssystems, so kann man auch beim Software-Prozeß Architektur und Implementierung unterscheiden: Das Prozeßmodell aus Abb. 2 stellt einen Ausschnitt aus der *Architektur* eines Prozesses dar. Die PE kann den Kontroll- und Datenfluß innerhalb des Prozesses anhand des Prozeßmodells steuern. Die Regeln, wie diese Steuerung funktioniert, stecken in der *Implementierung* der PE. Hier wird auf niedrigerer Abstraktionsebene und mit höherem Detaillierungsgrad beschrieben, welche konkreten Aktionen aufgrund von Änderungen des Prozeßzustands ausgeführt werden.

In *genform* ist die Implementierung des Prozesses in *Prozeßagenten* enthalten. Diese sind keine 'vollwertigen' Agenten im Sinne z.B. von [14]. Sie sind allerdings aktive und 'intelligente' Entitäten, die den Prozeß überwachen und selbständig Aktionen ausführen. Jeder Aufgabe ist ein *Task Execution Agent* (TEA) zugeordnet, der für die Ausführung der Aufgabe verantwortlich ist (Abb. 3) und ihren Kontext (Vorgänger, Nachfolger, Unteraufgaben, über Transitionen verbundene Aufgaben) kennt. Der TEA überwacht Zustandsänderungen des Prozesses und

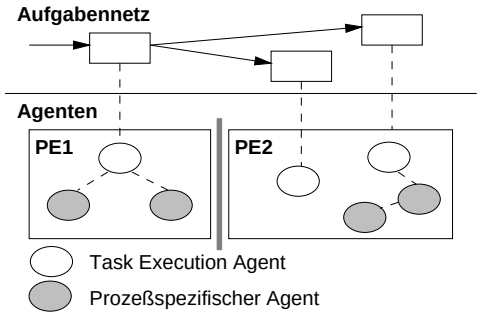


Abbildung 3. Aufgaben und Agenten

reagiert z.B. durch Änderung des Aufgabenzustands oder durch Weiterleiten eines Dokuments. Das Verhalten des Agenten wird dabei durch das Prozeßmodell vorgegeben (durch Beziehungen zwischen Aufgabentypen, Zustandsdiagramme

von Aufgaben und Dokumenten sowie Transitionen zwischen Ordnern). Durch Definition von Bedingungen und Aktionen innerhalb der Zustandsdiagramme kann der Prozeßmodellierer das Verhalten des TEA und damit den Ablauf des Prozesses beeinflussen.

Zusätzlich kann er *prozeßspezifische Agenten* implementieren und den Aufgabentypen zuordnen. Diese werden vom TEA geladen und aktiviert. Auch prozeßspezifische Agenten können weitere Agenten laden und kontrollieren; somit kann die Implementierung von Prozeßfragmenten (in Form von Agenten) modularisiert und in verschiedenen Prozessen wiederverwendet werden.

Mit den Prozeßagenten wird das Konzept der Werkzeugkomponenten auf die Prozeßmaschine übertragen: Komponenten aus dem *genform*-Framework können vom Prozeßmodellierer direkt verwendet und parametrisiert werden. Umfangreichere Anpassungen sind durch Erweiterung der Komponenten auf Programmiersprachenebene möglich.

5 Implementierung

genform [17, 24] ist in Java implementiert unter Verwendung der *Swing*-GUI-Bibliothek. Als Repository wird H-PCTE [16] eingesetzt, eine *h*auptspeicherbasierte und damit *h*ochperformante Implementierung großer Teile des ISO-Standards 13719 PCTE [13, 32]. H-PCTE enthält Erweiterungen gegenüber dem Standard, mit denen die Realisierung von Software-Entwicklungsumgebungen erleichtert wird, darunter einen verteilten Benachrichtigungsmechanismus [30] und ein Werkzeugtransaktionskonzept mit feingranularem Sperrmechanismus [29]. Java-Programme greifen über das Java-API *JHPcte* [18] auf das Repository zu.

6 Zusammenfassung und Ausblick

Im Papier haben wir einen Ansatz vorgestellt, mit dem CASE-Werkzeuge konstruiert und in einer PSEU integriert werden können. Somit können Werkzeuge an Konzepte, Notationen *und* das Vorgehensmodell der verwendeten Software-Entwicklungsmethode angepaßt werden. Durch Parametrisierung generischer Werkzeugkomponenten (mit dem Datenmodell der Dokumente und mit Werkzeugparametern) können Werkzeuge einfach und schnell (aber nur begrenzt) adaptiert werden. Durch Erweiterung von Framework-Komponenten steht dem Werkzeugentwickler eine hohe Flexibilität (bei größerem Einarbeitungs- und Implementierungsaufwand) zu Verfügung.

Mit den Konzepten, die für die Konstruktion von CASE-Werkzeugen eingesetzt werden, können auch Werkzeuge für die Modellierung, Planung und Kontrolle von Software-Prozessen einschließlich einer modularen Prozeßmaschine realisiert werden. Durch die resultierende enge Integration der Werkzeuge mit dem Prozeß wird eine feingranulare Prozeßunterstützung für Entwickler und Manager möglich.

Zentrale Architekturkomponente des beschriebenen Ansatzes ist das Repository, das Produkt-, Prozeß- und organisatorische Daten verwaltet, und dessen Dienste für die Implementierung der Funktionalität von Werkzeugen und Prozeßmaschine wie für deren Integration vorteilhaft genutzt werden.

Zukünftige Aufgaben umfassen die Identifikation und Implementierung häufig benötigter Prozeßagenten, den Ausbau der Werkzeugunterstützung für Planung und Management, die Verfeinerung der graphischen Darstellung komplexer Dokumente und die Erweiterung der Mechanismen zur Konsistenzsicherung und Reportgenerierung.

Literatur

[1] 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.

[2] V. Ambriola, R. Conradi, and A. Fuggetta. Assessing Process-Centered Software Engineering Environments. *ACM Transactions on Software Engineering Methodology*, 6(3), July 1997.

[3] H. Balzert. *Methoden der objektorientierten Systemanalyse.* BI Wissenschaftsverlag, Mannheim, 1995.

[4] D. Däberitz and U. Kelter. Rapid Prototyping of Graphical Editors in an Open SDE. In *Proceedings of the International Conference of Software Development Environments and CASE (SEE)*, pages 61–72, Noordwijkerhout, Netherlands, Apr. 1995. IEEE Press.

[5] 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 E. Bertino, editor, *Proc. 14^{th} European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 27–43, Sophia Antipolis and Cannes, France, June 2000. Springer.

[6] 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.

[7] 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. [8].

[8] J. Gray, J. Harvey, A. Liu, and L. Scott, editors. *Proc. 1st Int. Symposium on Constructing Software Engineering Tools (COSET '99)*, Los Angeles, CA, May 1999.

[9] 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.

[10] W. Harrison, H. Ossher, and P. Tarr. Software Engineering Tools and Environments: A Roadmap. In ICSE'00 [12], pages 261–277.

[11] 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.

[12] *Proceedings of the 22^{nd} Int. Conference on Software Engineering (ICSE'00)*, NY, June 4–11 2000. ACM Press.

[13] 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.

[14] N. R. Jennings and M. J. Wooldridge. Software Agents. *IEE Review*, pages 17–20, Jan. 1996.

[15] S. Kelly and K. Smolander. Evolution and issues in metaCASE. *Information and Software Technology*, 38(4):261–266, 1996.

[16] U. Kelter. H-PCTE – a High Performance Object Management System for System Development Environments. In *Proceedings of the 16th Annual International Computer Software and Application Conference (COMPSAC)*, pages 45–50, Chicago, Illinois, Sept. 1992. IEEE Computer Society Press.

[17] 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.

[18] U. Kelter, M. Monecke, and D. Platz. Constructing Distributed SDEs using an Active Repository. In Gray et al. [8].

[19] M. Koskinen and P. Marttiin. Developing a customizable process modelling environment: Lessons learnt and future prospects. *Lecture Notes in Computer Science*, 1487:13–27, 1998.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] B. Nuseibeh, A. Finkelstein, and J. Kramer, editors. *Software Process Modelling and Technology*. John Wiley and Sons, 1994.

[28] OMG. *OMG Unified Modeling Language Specification, Version 1.3*. Object Management Group (OMG), Mar. 2000.

[29] D. Platz. *Ein Werkzeugtransaktionskonzept für Objekt-Managementsysteme als Basis von Software-Entwicklungsumgebungen*. Shaker Verlag, Aachen, 1999.

[30] D. Platz and U. Kelter. Konsistenzerhaltung von Fensterinhalten in Software - Entwicklungsumgebungen. *Informatik Forschung und Entwicklung*, 12(4):196–205, Dezember 1997.

[31] I. Thomas and B. A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, 9(2):29–35, Mar. 1992.

[32] L. Wakeman and J. Jowett. *PCTE – The Standard for Open Repositories*. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1993.

[33] B. Westfechtel. *Models and Tools for Managing Development Processes*, volume 1646 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, 1999.