Graphische Unterstützung der Entwicklung verteilter Anwendungen

Torsten Leidig, Max Mühlhäuser
FB Informatik, AG Telematik
Universität Kaiserslautern
Erwin-Schrödinger-Str., D-6750 Kaiserslautern
[+49]-631-205-2803, leidig@informatik.uni-kl.de

Abstract:

Der ständig zunehmende Einsatz verteilter DV-Systeme führt zu einem stark steigenden Bedarf an *verteilten Anwendungen*. Deren Entwicklung in den verschiedensten Anwendungsfeldern wie Fabrik- und Büroautomatisierung ist für die Anwender bislang kaum zu handhaben. Neue Konzepte des Software Engineering sind daher notwendig, und zwar in den drei Bereichen ‘Sprachen’, ‘Werkzeuge’ und ‘Umgebungen’. Objekt-orientierte Methoden und graphische Unterstützung haben sich bei unseren Arbeiten als besonders tauglich herausgestellt, um in allen drei Bereichen deutliche Fortschritte zu erzielen. Entsprechend wurde ein *universeller objektorientierter graphischer Editor*, ***ODE***, als eines unserer zentralen Basis-Werkzeuge (‘tool building tool’) entwickelt. ODE basiert auf dem objekt-orientierten Paradigma sowie einer leicht handhabbaren funktionalen Sprache für Erweiterungen; außerdem erlaubt ODE die einfache Integration mit anderen Werkzeugen und imperativ programmierten Funktionen. ODE entstand als Teil von DOCASE, einer Software-Produktionsumgebung für verteilte Anwendungen. Grundzüge von DOCASE werden vorgestellt, Anforderungen an ODE abgeleitet. Dann wird ODE detaillierter beschrieben. Es folgt eine exemplarische Beschreibung *einer* Erweiterung von ODE, nämlich der für die DOCASE-Entwurfssprache.

1 Einführung

Schnelle Netze, integrierte Telematik-Dienste und Multimedia-Kommunikation bringen nicht nur neue Bewegung in die Welt der Kommunikationsprotokolle, sie lassen vor allem in der Anwendungsschicht den Ruf adäquater *Softwaretechnik für verteilte Anwendungen* noch lauter werden: in den Bereichen CIM und Büroautomation sind verteilte Systeme Stand der Technik, aber Anwender haben noch immer keine geeignete Hilfsmittel, um verteilte Anwendungen zu erstellen.

DOCASE, ein Projekt der Universitäten Kaiserslautern und Karlsruhe und des Digital Equipment CEC Forschungszentrums in Karlsruhe, hat zum Ziel, eine prototypische Softwareproduktionsumgebung für verteilte Anwendungen zu erstellen; die erste Version von DOCASE ist in der Implementierung weit fortgeschritten. Eines der zentralen Ziele von DOCASE war eine umfassende graphische Unterstützung auf der Basis des hier vorgestellten erweiterbaren objekt-orientierten graphischen Editors ODE. ODE soll in dieser Arbeit vorgestellt werden. In Kapitel 2 werden die wesentlichen Charakteristika der Software-Produktionsumgebung DOCASE vorgestellt; daraus werden Anforderungen an den Graphikeditor ODE abgeleitet. In Kapitel 3 wird ODE dann detailliert dargestellt. Kapitel 4 verdeutlicht die Erweiterbarkeit von ODE und beschreibt die Anpassung des Graphikeditors an die in DOCASE verwendete objekt-orientierte Entwurfssprache DODL (DOCASE Design Langugage)[1, 2].

2 DOCASE Grobarchitektur

Abb. 2-1 zeigt grob die wichtigsten Funktionsblöcke der Softwareproduktionsumgebung.

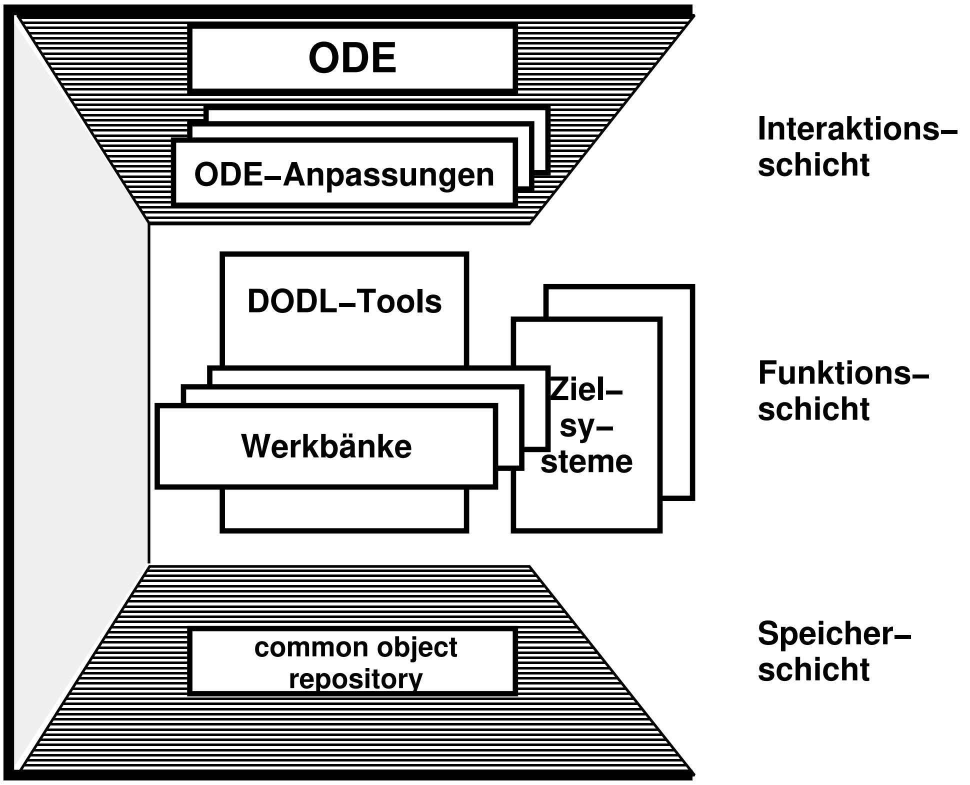

Abbildung 2-1 DOCASE und ODE

DODL-Tools: Der sematische Kern von DOCASE ist die objekt-orientierte Weitbereichs-Sprache DODL (DOCASE Design-oriented Language). Sie zeichnet sich im wesentlichen aus durch

- spezielle Unterstützung der Besonderheiten verteilter Anwendungen (Komplexität, Parallelität, Dynamik und Laufzeit, Asynchronität...)

- Verwendung des objekt-orientierten Paradigmas, welches sich für verteilte Programmiersprachen zunehmend durchsetzt aufgrund seiner besonderen Eignung für strukturierte netztransparente Programmierung

- Überwindung wesentlicher Nachteile des objekt-orientierten Paradigmas: durch Anbieten eines hierarchischen Strukturierungskonzeptes (welches anders als die übliche Vererbungshierarchie nicht "verwandte" sondern "zusammengehörende" Objekte zu gruppieren gestattet), sowie durch Festlegen einer allgemein akzeptierten, allen Werkzeugen bekannten Grundtypisierung von Objekten ("Subsysteme", "Ablaufobjekte", "dynamische "Fluß"-Objekte", "semantische Relationen" etc.)

- Weitbereichscharakteristik: Tauglichkeit als Entwurfs- *und* Implementierungsprache. Als Entwurfssprache durch Abstraktionen, Zulassen unvollständiger Spezifikationen, *graphische Repräsentation* (s. Kap. 4). Als implementierungsnahe Sprache durch Vollständigen Sprachumfang für verteilte Programmierung.

Es wird davon ausgegangen, daß sich auf verschiedenen Zielumgebungen verschiedene (verteilte objekt-orientierte) Programmiersprachen durchsetzen werden. DODL und DOCASE sind so angelegt, daß unterschiedliche derartige Zielumgebungen eingebunden werden können. Von einer implementierungsnahen DODL-Repräsentation wird also i.a. nicht in Maschinencode, sondern in Zielsprach-Code übersetzt. Als erstes wird eine verteilte Version Trellis [3] angeboten.

Die wichtigsten Werkzeuge zur Verarbeitung von DODL sind

- in der Interaktions-Schicht: die DODL-Erweiterung von ODE (s. Kap. 4)

- in der Funktionsschicht: Übersetzer, Interpretierer und Animationswerkzeug (dieses ist eng mit dem Graphikwerkzeug gekoppelt).

Werkbänke: Ein wichtiger Teil von DOCASE ist die Extraktion und Integration von Entwicklungsaspekten. Die Komplexität verteilter Anwendungen macht es erforderlich, eine Vielzahl von Aspekten, wie 'Leistungsoptimierung mittels Simulation', 'Kommunikationskostenoptimierung mittels Objekt-Migrations-Heuristiken', 'Einbringen von Fehlertoleranz' u.v.m. detailliert zu betrachten. Zwei Probleme erschwerten traditionell die Betrachtung solcher Aspekte:

A. Die Aspekt-spezifischen Teile der verteilten Anwendung waren oft im Entwurf und Quellcode mit dem eigentlichen funktionalen Entwurf völlig durchmischt (z.B. Konfiguration, Fehlertoleranz); eine isolierte Betrachtung eines Aspekte oder der funktionalen Elemente der Anwendung war nicht möglich. Oder aber die Aspekte wurde völlig separat behandelt (z.B. simulative Leistungsbewertung) und waren fast nicht mit den laufenden Veränderungen der in Entwicklung befindlichen Anwendung konform zu halten.

B. Bei vielen Aspekten existierten verschiedene Werkzeuge und damit oft verschiedene Modelle, Sprachen etc. für verschiedene Phasen des Software-Lebenszyklus (z.B. Formulierung interessierender Leistungshemmgrößen oder operationaler Randbedingungen bei der simulativen Modellierung: in der Entwurfsphase oft völlig anders als bei der direkten Leistungsmessung in der Betriebsphase).

Um diese Nachteile zu beseitigen, wurde in DOCASE das Werkbank-Konzept eingeführt. Eine Werkbank

- besteht aus einem Satz von Werkzeugen für denselben Aspekt
- wird über eine einheitliche (graphische) Oberfläche bedient
- isoliert einerseits die aspektbezogenen Anforderungen an die verteilte Anwendung, bringt sie andererseits mit den Anforderungen anderer Aspekte sowie mit dem funktionalen Kern der Anwendung in Beziehung; dies geschieht unter Ausnutzung besonderer Strukturierungshilfsmittel der Sprache DODL (insbesondere der "semantischen Relationen", welchen für jeden Aspekt einige Klassen zur Verfügung stellen).

Neue Werkbänke werden eingebracht, indem die Werkzeuge entwickelt werden, mittels ODE die graphische Oberfläche implementiert wird, DODL - insbesondere um geeignete Klassen semantischer Relationen - erweitert wird und ggf. die Laufzeitumgebungen angepaßt werden (z.B. über Feedback von Monitoring-Daten).

DOCASE-Schale: diese enthält zum einen eine Portierungs-Schnittstelle mit der in der Interaktionsschicht an verschiedene Benutzerschnittstellen-Standards (derzeit X-Windows), in der funktionalen Schicht an verschiedene Betriebssysteme und Netzwerkarchitekturen (derzeit UNIX und TCP/IP) und in der Speicherschicht an verschiedene Datenverwaltungssysteme (s.u.) angepaßt weren kann. Zum anderen enthält sie Tool-building-Tools wie ODE, aus denen andere Werkzeuge generiert bzw. die an unterschiedliche Bedürfnisse angepaßt werden können.

Common object repository: in der Speicherschicht wird durch die Abstützung eines Standards für die Speicherung von Entwicklungs-Artifakten ("CATIS", beinhaltet gleichzeitig Modelle für Ressourcen- und Code/Versionmanagement) Portierbarkeit erreicht.

3 ODE: ein universeller objektorientierter graphischer Editor

3.1 Anforderungen an den graphischen Editor

Die Anforderungen lassen sich grob in verschiedene Gruppen einteilen. Zunächst sind die Anforderungen an den graphischen Editor als *Tool* relevant:

- Erweiterbarkeit
- Flexibilität

- Ausdrucksstärke

Wie im vorherigen Kapitel dargelegt, wird im Projekt DOCASE versucht, der Komplexität der Entwicklung verteilter Anwendungen durch die klare Abgrenzung eines Spektrums von Aspekten zu begegnen sowie durch die Entwicklung entsprechender 'Werkbänke'. Darüberhinaus sollen die Entwicklungsschritte grafisch unterstützt werden, wobei eine möglichst einheitliche graphische Oberfläche sehr wichtig ist. Ein zentraler Ansatz hierbei ist der, ein generisches Werkzeug zur Verfügung zu stellen, mit dem die spezifischen graphischen Werkzeuge für die Werkbänke erstellt werden. Für derartige Werkzeuge hat sich der Term 'tool building tool' eingebürgert. Mit diesem Ansatz wird eine einheitliche Oberfläche geschaffen und darüberhinaus die Integration von Werkzeugen erleichtert. Jedoch muß der zugrunde liegende generische Editor flexibel und mächtig genug sein, um für alle gewünschten Erweiterungen tauglich zu sein.

Eine zweite Menge von Anforderungen betrifft den *Software-Engineering Aspekt:*

- Unterstützung von speziellen Entwurfsmethoden, z.B. 'Schrittweise Verfeinerung'
- Zulassen von unvollständiger Spezifikationen
- Unterstützung hierarchischer Strukturierung
- Lokalität von wesentlichen Informationen, dargeboten in speziellen Sichten auf das System

Speziell die erste Anforderung bedeutet insbesondere, daß Methoden vom Ersteller vorgegeben werden können, nach denen der Benutzer des Editors sich in bestimmten Grenzen richten muß. Dies geht über die übliche Unterstützung von einem bestimmten methodischem Vorgehen wesentlich hinaus. Ein (schlechtes) Beispiel wäre etwa die Erzwingung von Top-down-Entwurf.

Der letzte Punkt berührt eine wichtige Grundidee des Editors: die Zentralisierung und ggf. Isolation von Informationen zu einem bestimmten Aspekt der Entwicklung verteilter Anwendungen. Als Beispiel sei hier der Aspekt "Kommunikation" genannt, durch dessen Zentralisierung es möglich wird, *Muster und Abläufe* von Komunikationsbeziehungen zwischen Objekten an einer zentralen Stelle zu beschreiben, anstatt - wie heute üblich - die Kommunikation zwischen Objekten (in Form von in den Objektcode eingestreuten Datenaustauschoperationen) über alle beteiligten Objekte zu 'verwischen'.

Der Begriff *Lokalität* bedeuted in diesem Zusammenhang, daß alle zur Untersuchung eines Aspektes benötigten Eigenschaften (Daten) möglichst zentral vorliegen sollten und nicht über das gesamte System verstreut oder gar nur implizit vorhanden. Für den graphischen Editor heißt das, daß es möglich sein muß, solche Daten zu Aspekten zusammenzuführen, die auch in einem graphischen Zusammenhang dargestellt werden können (etwa in einem Fenster, auch 'Sicht' genannt). Die Darstellungsattribute der Objekte und Eigenschaften in einer speziellen Sicht sind dabei aspektspezifisch.

Die unreflektierte Entgegennahme eingegebener Informationen wird für interaktives graphische Werkzeuge schon lange Zeit nicht mehr als adäquat betrachtet. Üblich ist hier Computerunterstützung, d.h. das Werkzeug prüft Eingabedaten und leitet ggf. selbstständig resultierende Daten ab. Außerdem sollten über das gesamte System semantische Analysen - z.B. Konsistenz- und Vollständigkeitsüberprüfungen als auch domänenspezifische Analysen - durchgeführt werden können. Wir spezifizieren daher folgende detailliertere Anforderungen:

- Syntax- und Semantik- gesteuertes Editieren
- Semantische Analysen
- Animation

Animation steht hier für die Visualisierung von Abläufen im System. Sie ist nützlich, um dem Entwickler eine bessere Vorstellung und ein besseres Verständnis für das komplexe Systemverhalten zu geben. Dabei sind eine Vielzahl verschiedener interessierender Parameter vorstellbar, die sich in einer grafischen Repräsentation im Editor widerspiegeln.

Speziell für die DOCASE Umgebung, in der Objektbeziehungen eine wesentliche Rolle spielen, ist deren adäquate Darstellung notwendig. Aus diesem Grund wurde für den graphischen Editor ein Schwergewicht auf gerichtete Graphen gelegt.

3.2 State of the Art

In diesem Abschnitt soll kurz der aktuelle Stand der Forschung auf dem Gebiet der graphischen Tools umrissen werden. Mit dem Aufkommen komfortabler graphischer Benutzeroberflächen für die standard ‘Desktop’-Systeme (Arbeitsstationen, z.T. auch PCs mit Fenster-Oberflächen) wurden vermehrt graphische Tools im Software Engineering Bereich entwickelt. Die Benutzeroberfläche von Smalltalk [4, 5] ist Vorbild vieler nachfolgender Fenstersysteme: eines der jüngsten und vielversprechendsten Beispiele ist das X-Window System [6]. Im wesentlichen sind deren Eigenschaften und Mechanismen ähnlich. Im folgenden soll daher die besondere Problematik von graphischen Editoren und Programmiersprachen stehen.

Durch die parallele Darstellung in mehreren ‘Fenstern’ entstand insbesondere das Problem der Teilung gemeinsamer interner Daten und der Konsistenz der Darstellung (z.B. müssen alle Instanzen einer Klasse ’ClassBrowser’ müssen eine interne Veränderung der Klassenhierarchie in ihrer Darstellung berücksichtigen). Das Smalltalksystem behandelt derartige Abhängigkeiten mit Hilfe des Model-View-Controller (MVC) Ansatzes.

Ein anderer Ansatz zur Behandlung dieser Problematik ist der ’Daemon’-Ansatz [7]. Daemons sind Prozesse im Hintergrund, die bestimmte Aktivitäten des Systems überwachen. Bei auftreten eines bestimmten Ereignisses ’feuern’ sie, d.h. sie werden aktiv und führen ihrerseits eine Reihe von Aktionen aus um das ganze System zu aktualisieren.

In dem ‘Frame’-basierten System KEE [8] werden sogenannte ’active values’ eingesetzt. Wann immer auf den Wert eines ‘Slots’ (Eintrag in einem ‘Frame’) zugegriffen wird oder der Slot anderweitig aktiviert wird, wird eine benutzerdefinierte Aktion gestartet. Bemerkenswert ist, daß aus dieser Technik ein neuer Programmierstil entstanden ist (‘data driven programming’). Dabei werden Methoden (=Funktionen) ausgeführt, wenn bestimmte Datenfelder geändert werden. In KEE werden über diese Technik z.B. auch Regeln eines Regelinterpreters getriggert.

Eine weitere Formalisierung der Behandlung von Objektabhängigkeiten bringen die ’constraint-based’ Sprachen bzw. Systeme. Constraints sind Abhängigkeiten zwischen Objekten, welche zu erfüllen sind. Man kann den Begriff Constraint als ‘Randbedingung’ auffassen. Die Einhaltung der Constraints zu überwachen (’constraint satisfaction’) obliegt dem sog. ‘Constraint-Resolver’. Es wird unterschieden zwischen unidirektionalen Constraints, bei denen die Abhängikeit eines Wertes von anderen Werten (nur in einer Richtung) angegeben wird, und ‘mehrdirektionalen’ Constraints, welche wechselseitige Abhängigkeiten ausdrücken. Auch muß es nicht immer eine eindeutige Lösung der Constraints geben. Je nach Art der Constraints sind keine, eine, mehrere oder unendlich viele Lösungen möglich. Der Vorteil bei der Verwendung von Constraints liegt darin begründet, daß der Algorithmus zur Auflösung der Constraints unabhängig von den Constraints ist (etwa ’dependency-directed backtracking’ [9] oder Waltzalgorithmus [10]) und von einer abstrakten Maschine durchgeführt werden kann. Der Programmierer kann sich theoretisch auf die Formulierung der Constraints beschränken. Ein recht beeindruckendes graphisches Constraint-System ist ThingLab [11, 12] von Alan Borning. ThingLab gestattet die graphische Spezifikation von Constraints. Die Ergebnisse der Constraint-Auswertung werden direkt auf dem Bildschirm dargestellt. Die Auswertung eines komplexen nichtsymbolischen Constraint-Netzwerkes ist sehr rechenintensiv, so daß die Zeitanforderungen für interaktive Systeme nicht immer erfüllt werden können. Die Verwendbarkeit hängt wesentlich von der schnellen Auswertung des Constraint-Netzwerkes ab. Andererseits stellen Constraints einen uniformen, mächtigen und geschlossenen Ansatz zur Behandlung von Objektabhängigkeiten dar. Neuere visuelle Programmierumgebungen wie z.B. ’ThinkPad’ [13], ’Fabrik’ [14] und ’Rehearsal World’ [15] wurden stark von ThingLab beeinflußt.

3.3 Konzepte von ODE

Nachfolgend sollen die zentralen Konzepte des ODE-Editors vorgestellt werden, mit deren Hilfe versucht wurde, die in Abschnitt 3.1 genannten Anforderungen so weit wie möglich zu reflektieren.

3.3.1 Datenmodell

Eine zentrale Eigenschaft von ODE ist seine Erweiterbarkeit. Die beiden wesentlichen Ansätze, mit denen diese Erweiterbarkeit in ODE erreicht wird, sind ein *einfach zu handhabendes internes Datenmodell* sowie eine *eingebettete Lisp-ähnliche funktionale Spra-*

che. Letztere enthält die üblichen Datentypen wie z.B. Integer, Character, Boolean, String, Real, Symbole und als zusammengesetzte Datentypen Listen und Felder. Daneben bietet sie als universelle Erweiterung einen Objekt-Typ an, der als Basis-Baustein für objektorientierte Techniken dient. Das Objekt-Modell ist elementar, d.h. die Basiseigenschaften aller üblichen objektorientierten Systeme sind erfüllt und eine Anpassung an die Besonderheiten individueller Systeme ist möglich. Die Lisp-ähnliche *funktionale Sprache* dient aussschließlich der 'tool-building'-Funktion von ODE, d.h. der Anpassung ('customization'), mit der ein spezielle Editor für eine spezielle Entwurfsmethode oder Technik aus ODE erzeugt wird. Für den Endbenutzer (des angepaßten Editors) ist diese Ebene nicht mehr sichtbar, so daß er insbesondere die funktionale Sprache *nicht beherrschen muß!*

Mit dem Einsatz einer funktionalen Programmiersprache werden die folgenden, konkreten Ziele verfolgt:

- Die Sprache dient der *Modellierung* der Struktur (Daten) und des funktionalen Verhaltens eines Systems abhängig von der spezifischen Anwendung.
- Sie ermöglicht die Programmierung extensiver *semantischer Überprüfungen* der Eingabe.
- Sie ermöglicht die Implementierung von *Simulationen* und *Animationen* des Modells.

3.3.2 Objektmodell

Bei der Entwicklung des ODE-Objektmodelles wurden zwei wesentliche Leitlinien verfolgt: zum einen sollte der Objekttyp an die Bedürfnisse des graphischen Editors angepaßt sein; zum anderen wurde wie erwähnt versucht, einen elementaren Objekttyp zu schaffen, der jedoch zu verschiedenen objektorientierte Techniken, wie sie z.B. auch im Softwareentwurf zum Einsatz kommen, erweitert werden kann. Letzterer Aspekt hat wiederum Einfluß auf die Erweiterbarkeit des graphischen Editors.

Ein Objekt in ODE ist gekennzeichet durch seine Klasse und die daraus resultierenden Eigenschaften, sowie durch seine Beziehungen zu anderen Objekten. Ein Objekt gehört genau einer Klasse an. Die Eigenschaften werden durch Symbole benannt und können von einem beliebigen Typ sein. Klassen sind spezielle Objekte, die das Verhalten einer Objektklasse festlegen; diese Verhaltensbeschreibung erfolgt durch die Angabe der gemeinsamen Eigenschaften jeder Instanz der Klasse, sowie der für die Klasse gültigen Operationen. Klassen stehen durch eine Vererbungshierarchie zueinander in Beziehung, 'multiple inheritance' ist hierbei zulässig. Die Vererbungshierarchie wird über einen speziellen Beziehungstyp 'is-subclass' realisert.

3.3.3 Objektbeziehungen

Beziehungen zwischen Objekten werden als Objekte (Instanzen) einer speziellen Beziehungsklasse realisiert, die ihrerseits in einer Vererbungshierarchie steht. Es ist damit eine

Typisierung von Beziehungen möglich. Durch die Instanzenvariablen von Beziehungen können diese attributiert werden, durch die Methoden der Beziehungsklasse bekommen sie eine operationale Semantik.

Beziehungen sind in ODE Primitive der Sprache. Diese Vorgehensweise hat eine Reihe von Vorteilen, wie bereits andere Autoren [16, 17] feststellten:

- mächtige uniforme Funktionen über Relationen, wie etwa Hüllenbildung, Traversierungsstategien, usw. können unabhängig von der speziellen Semantik der Relation angewendet werden;
- Beziehungen machen Eigenschaften des Systems explizit, die sonst im Programmcode 'versteckt' sind;
- speziell für graphische Editoren ist von Bedeutung, daß Graphen, welche von Objektbeziehungen aufgespannt werden, standardmäßig durch verschiedene Graph-Layout-Algorithmen aufbereitet und dargestellt werden können.

Wie im vorangegangenen Abschnitt erwähnt, ist die Klassenhierarchie mit Hilfe des Beziehungstyps 'is-subclass' realisiert. Dies ist ein Beispiel für das zentrale Konzept von ODE, mit wenigen aber mächtigen Primitiven auszukommen. Dies führt zu einer Selbstbezüglichkeit, welche bei ODE beabsichtigt ist. Diese Eigenschaft wird auch mit *'self-contained'* oder *'reflective'* bezeichnet. Sie hat den Vorteil, daß man mit den Funktionen und Tools des Editors den Editor selbst verändern kann. Bezogen auf die Klassenhierarchie bedeutet das z.B., daß man durch Veränderung der 'is-subclass'-Klasse eine völlig andere Vererbungssemantik z.B. in Bezug zur Mehrfachvererbung erzielen kann. Die Klassenhierarchie besitzt z.B. in der generischen Version von ODE nur eine Metaklasse, nämlich die Klasse 'Class', die eine Instanz von sich selbst ist. Smalltalk kennt dagegen eine zur eigentlichen Klassenhierarchie parallele Metaklassenhierarchie. Durch eine Veränderung der 'is-subclass'-Methoden kann der ODE-Programmierer nun auf einfache Weise ein Smalltalk-ähnliches Verhalten erzeugen, wenn seine Anwendung das erfordert. Darüberhinaus kann man zur Änderung der Klassenhierarchie die entsprechenden ODE-Tools für Objektbeziehungen (also z.B. den Grapheneditor) unverändert benutzen.

3.3.4 Abbildung Datenmodell in graphische Repräsentationen

Eines der Hauptprobleme graphischer Editoren liegt in der Auswahl einer geeigneten *visuellen Repräsentation*. In der 2-dimensionalen Ebene heutiger Ausgabegeräte, lassen sich eine Reihe von Größen und Qualitäten angeben, die für die Kodierung von Informationen genutzt werden können, wie z.B. x/y-Position, absolute/relative Größe, Dicke, Ausdehnung, Linienarten, Formen, räumliche Beziehungen, Symbolik etc.. Gute graphische Techniken des Entwurf nutzen aber meistens nur eine beschränkte Auswahl dieser Größen. Dennoch werden aber bei verschiedenen Techniken sehr verschiedene graphische Möglichkeiten genutzt, sodaß ein generisches Werkzeug wie ODE ein verallgemeinertes Modell der Abbildungstechniken anbieten sollte.

Ein wichtiges Prinzip in ODE ist die *explizite* Unterscheidung von Modell und graphischer Repräsentation (s. Abb. 3-1), ähnlich dem MVC-Paradigma in Smalltalk. Der Sinn dieser Trennung liegt in den folgenden Punkten begründet:

- Ein Objekt des Modells kann mehrere und je nach dem Kontext in dem es erscheint verschiedene graphische Repräsentationen besitzen.
- Das Modell wird unabhängig von den Views und den darin enthaltenen grafischen Repräsentationen entworfen.
- Views mit verschiedenen graphischen Repräsentationen können flexibel auf beliebige Teile des Modells erzeugt werden.

ODE bedient sich der Abstraktion von graphischen Dialogobjekten, wie sie im X-Window-System und seinen Toolkits (z.B. OSF-Motif) unter der Bezeichung *Widget* Verwendung findet. Im Gegensatz zu den Primitiven auf der untersten X-Window-Ebene (Fenster, Linien, Schrift) sind Widgets graphische Objekte mit einer vorgegebenen Bedeutung und einem vorgegebenen Verhalten gegenüber dem Benutzer. Widgets sind z.B. Buttons, ListBoxes, editierbare Textfelder, Rollbalken, Schieberegler, also Dialogelemente auf einer höheren Abstraktionsstufe. Ihr visuelles Erscheinungsbild und ihr Verhalten ist im wesentlichen festgelegt und kann in den verschiedenen Instanzen parametrisiert werden.

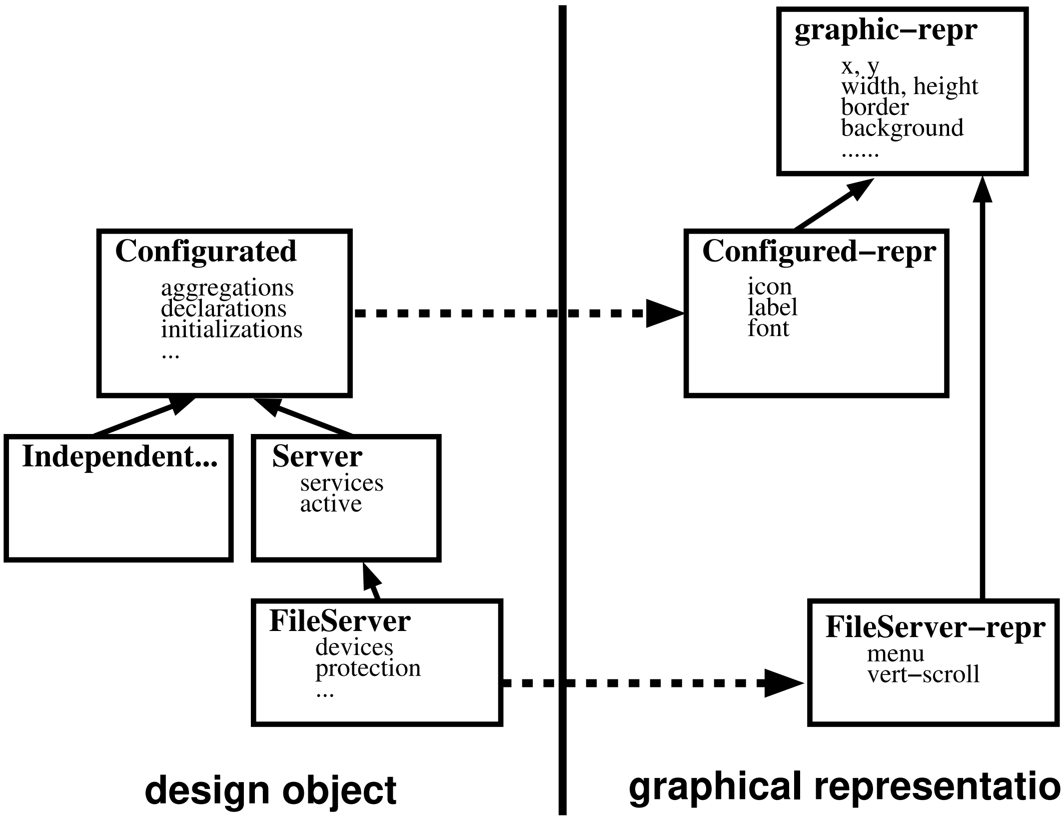

Abbildung 3-1 Beziehung zwischen Modell und graph. Repräsentation

3.3.5 Views

Der Begriff 'View' hat in ODE seine eigene Bedeutung. Ein View ist eine Sicht auf einen bestimmten Ausschnitt des Systems unter einem bestimmten Modellaspekt. Ein *atomarer* View ist eine graphische Repräsentation *eines* ODE-Datenobjektes, erweitert um seine spezifische Reaktion auf Benutzeraktionen. Ein *zusammengesetzer* View besteht dagegen aus mehreren graphischen Repräsentationen von ODE-Datenobjekten, z.B. eine Liste von Objekten. Mehrere Views können in einem Fenster miteinander kombiniert werden.

ODE bietet ein Kollektion von verschiedenen View-Typen, die je nach Typ der darzustellenden Daten eingesetzt werden können. Als atomare Views stehen u.a. editierbare Textfelder und Buttons zur Verfügung. Als Vertreter von zusammengesetzten Views ist der *List-View* zu nennen, der eine Liste von ODE-Datenobjekten zeigt.

Der zentrale zusammengesetzte View ist der *Graph-View*. Dieser View kann den Graphen, der durch die Beziehungen zwischen Objekten aufgespannt wird, darstellen. Objekte im Graphen werden durch rechteckige Kästen (Boxen) graphisch repräsentiert. Die Boxen enthalten einen Label und ein optionales Icon. Die Beziehungen zwischen Objekten werden graphisch als Kanten zwischen den Objektrepräsentationen dargestellt.

Die dargestellten Beziehungen lassen sich vom Typ her einschränken. Man kann sich auf einen Beziehungtyp beschränken oder beliebig viele Beziehungstypen zulassen. Der Graphenumbruch (Layout) geschieht dabei automatisch, wobei verschiedene Algorithmen angeboten werden. Das Layoutprogramm ist eine eigenständige Arbeit [18], die hier in ODE als Modul eingebunden wurde. Die Layoutalgorithmen sind austauschbar, es lassen sich jederzeit neue Algorithmen implementieren.

Der Graph-View ist darüber hinaus in der Lage einen konkreten Beziehungstyp als Abstraktionsbeziehung zu benutzen, d.h. Beziehungen von diesem Typ zwischen Objekten werden nicht in der üblichen Weise als Kante gezeichnet sondern zur Bildung von Untergraphen verwendet.

3.3.6 Behandlung von Constraints

Die Abhängigkeiten zwischen graphischen Repräsentationen und Teilen des Modells wurden bereits in Abschnitt 3.2 erläutert. In ODE wurde zunächst der Active-Value-Ansatz ähnlich zu KEE verfolgt. Dieser Ansatz hat den Vorteil schnell zu sein, da die Lösungsstrategie vom Programmierer im Einzelfall vorgegeben wird und daher optimiert werden kann. (Die Antwortzeiten spielen bei einem interaktiven Editor eine große Rolle, so sind Antwortzeiten von wenigen Sekunden oft schon nicht mehr akzeptabel.) Der Nachteil liegt jedoch darin, daß die Programmierung der Lösung in dieser Art recht kompliziert werden kann und deshalb auch fehleranfälliger ist. Für die Zukunft wird zusätzlich eine mehr deklarative Spezifizierung von Constraints und die Bereitstellung verschiedener allgemeiner Lösungsstrategien wie etwa 'dependency-directed backtracking' angestrebt. Ein weiterer Ansatz der verfolgt wird, ist die Formulierung von Constraints in Form von Regeln und die Verwendung eines Regelinterpretierers zur Auflösung der Constraints. Abschließend ist zu bemerken, daß die Active-Values bzw. Constraints nicht nur zur Behandlung der Abhängigkeiten zwischen Modell und graphischer Repräsentation genutzt werden können, sondern auch zur Beschreibung von Abhängigkeiten innerhalb des Modells. Davon wird in der DODL-Anwendung auch Gebrauch gemacht.

4 Die Anpassung von ODE an die Sprache DODL

Die Anwendung von ODE als Entwurfseditor für die Sprache DODL wird im folgenden mit *DODE* bezeichnet.

4.1 Anwendungsgebiet 'Visuelle Entwurfssprachen'

Diese spezielle Aufgabe des graphischen Editierens von Programmen geht in die Richtung der visuellen Programmierung, wie sie in Abschnitt 3.2 dargestellt wurde.

Während Programme in textuellen Sprachen als eine sequentielle Aneinanderreihung von Wörtern der Sprache zu verstehen sind, sind Programme visueller Sprachen eine Anordnung von graphischen Repräsentationen von Programmelementen auf einer Darstellungsfläche. Die räumliche Anordnung und die Verbindung von graphischen Objekten ist dabei Träger von syntaktischer und semantischer Information.

DODL ist eine objekt-orientierte Entwurfssprache mit einer Typhierarchie, generischen Typen, Vererbung, Methoden, Beziehungstypen und anderen Elementen die aus objekt-orientierten Sprachen bekannt sind. Es gibt eine Reihe von Vorschlägen für graphische Notationen im objekt-orientierten Entwurf. Der Prozeß der Bildung von etablierteKonventionen ist jedoch keineswegs abgeschlossen. Bekannte Techniken sind beispielsweise OOSD [19, 20], HOOD [21], Boochs Diagramme [22] und OMT [23]. Diese Techniken werden in diesem Artikel nur in sofern angesprochen, indem sie in DODE Eingang fanden.

Wir entwarfen eine Kollektion von experimentellen Diagrammtechniken für DODE, die in einigen Punkten an obengenannte Techniken angelehnt sind. Jeder einzelne Diagrammtyp von DODE bezieht sich auf einen bestimmten Aspekt des Entwurfs verteilter Anwendungen.

4.2 Diagramme in DODE

Die DODE-Bezeichnungen für die einzelnen Diagramme heißen *Typendiagramm*, *Objektdiagramm, Methodendiagramm und Konfigurationsdiagramm.*

Die graphische Repräsentation von DODL Sprachelementen besteht zunächst aus einem Satz von Icons für die verschiedenen generischen Typen. Allerdings soll es auch möglich sein, in Erweiterungen von DODL (etwa auf den Bereich Büroautomatisierung), die sich durch Spezialisierungen der generischen Typhierarchie äußern, jederzeit neue Icons einzuführen. Die mehrstelligen Beziehungen, die DODL anbietet, werden ebenfalls durch Icons dargestellt. Ein Diagrammtyp wird dann durch entsprechende Views realisiert.

Das *Typendiagramm* zeigt die Typhierarchie der DODL-Typen. Der Benutzer kann mit Hilfe des Diagrammes die Hierarchie anschauen und verändern. Der Graph der Typhierarchie ist hierarchisch und frei von Zyklen, somit ein Idealfall für den Graphenumbruchsalgorithmus von ODE. Das Typendiagramm kann darüberhinaus dazu benutzt werden, um andere Beziehungen zwischen Typen als die der Subtypbeziehung zu beschreiben. Das Typendiagramm von DODE hat somit eine ähnliche Funktion wie das 'class diagram' von Booch [22], verwendet jedoch eine andere Darstellung der Grundelemente.

Die Objekttypen werden durch das *Objektdiagramm* (alternativ auch mit Matrixdiagramm bezeichnet) definiert. Es zeigt die Instanzenvariablen des Objekttypes sowie die Operationen (Methoden) des Objekttyps, wobei die nach außen sichtbaren Operationen hervorgehoben sind. Außerdem zeigt es auch die Komposition des Objekttyps aus Teiltypen. Diese Kompositonsbeziehung bildet eine baumartige Hierarchie, die im Objektdiagramm als Graph-Untergraph-Hierarchie dargestellt wird, in die der Benutzer beliebig ab- und auftauchen kann (zoomen).

Abbildung 4-1 Beispiele für DODE Diagramme

Das *Methodendiagramm* wird benutzt, um den Ablauf innerhalb einer Methode zu definieren. Die Anweisungen sind als eine Mischform zwischen Kontrollflußdiagramm und Strukturdiagramm dargestellt. Das *Konfigurationsdiagramm* dient zum Erfassen der initialen Konfiguration der Objekte der verteilten Anwendung. In diesem Diagramm sind reale Instanzen von Typen der Gegenstand der Betrachtung.

Mehr spezialisierte Diagramme, die auf bestimmte Aspekte der verteilten Anwendungssystementwicklung eingeht, sind in Entwicklung. So z.B. das Kolokationsdiagramm, das zur Beschreibung von generischen *Kolokationen* von Objekten (Kolokationsdefinitionen) dient. Eine (temporäre) Kolokation von Objekten auf einem Knoten des Rechnernetztes kann sinnvoll sein, um z.B. die Kommunikationskosten zu minimieren, da lokale Kommunikation i.A. weniger Zeit in Anspruch nimmt als Entfernte und außerdem keine Belastung für das Netz darstellt. Die Erzeugung von Kolokationen wird durch *Objektmigrationen* erreicht, bei denen wiederum Kosten zu berücksichtigen sind. Nicht alle Kollokationen sind auch technisch machbar, es kann hierbei zu Konflikten kommen. Ein graphisches Kolokationsdiagramm kann helfen, die Kolokationen zu definieren und ihre Beziehungen untereinander zu analysieren. Es kann auch die aktuellen Kolokationen in einem bestimmten Zustand der Simulation graphisch anzeigen.

4.3 Praktische Erfahrungen

Die Entscheidung, ein graphisches Editortool zu entwickeln, hat sich für die DOCASE-Umgebung bezahlt gemacht. Es konnten in kurzer Zeit einige spezielle Prototypen von graphischen Editoren erstellt werden, die sonst wesentlich längere Entwicklungszeiten erfordert hätten. Natürlich wurden bei der Entwicklung der Editoren einige Schwächen von ODE festgestellt, die jedoch die Verwendbarkeit für den speziellen Zweck nicht grundsätzlich in Frage stellten. Vielmehr werden die Erfahrungen zur weiteren Verbesserung von ODE genutzt, dessen Entwicklung beileibe noch nicht abgeschlossen ist.

5 Literatur

[1] W. Gerteis, A. Schill, L. Heuser, M. Mühlhäuser, *"DODL: A Design Language for Distributed Object-Oriented Applications"*, unpublished, University of Karlsruhe, Institute of Telematics

[2] A. Schill, L. Heuser, M. Mühlhäuser, "Using the Object Paradigm for Distributed Application Development", In: P.J. Kühn (Hrsg.), *"Kommunikation in verteilten Systemen"*, Proceedings : ITG/GTI-Fachtagung, Grundlagen, Anwendungen, Betrieb, Stuttgart, Feb 1989, Springer-Verlag

[3] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, "An Introduction to Trellis/Owl", *OPSLA'86 Proceedings*, ACM, Oct. 1986

[4] A. Goldberg, *"Smalltalk-80: The Interactive Language Environment"*, Addison-Wesley, 1984

[5] A. Goldberg, D. Robson, *"Smalltalk-80: The Language and its Implementation"*, Addison-Wesley, 1983

[6] R.W. Scheifler, J. Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, Apr. 1987, pp. 79-109

[7] Makoto Murata and Koji Kusumoto, "Daemon: Another Way of Invoking Methods", *JOOP*, Jul/Aug 1989.

[8] Renate Kempt, "Teaching object-oriented programming with the KEE system", *ACM SIGPLAN Notice*, 22(12), pp. 11-25, Dec. 1987

[9] R. Stallman and G.J. Sussman, "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis, *Artificial Intelligence,* 9(2), 1977

[10] D. Waltz, "Understanding Line Drawings of Scenes with Shadows", In: Patrick H. Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975

[11] A. Borning, "Defining Constraints Graphically", *Proc. CHI 86*, Conf. Human Factors in Computing Systems, Apr. 86, ACM, pp. 137-143.

[12] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, M. Woolf, "Constraint Hierarchies", *OPSLA'87 Proceedings*, ACM, pp. 48-60

[13] R.V. Rubin, E.J. Golin, and S.P. Reiss, "ThinkPad: A Graphical System for Programming by Demonstration", *IEEE Software*, Vol. 2, No. 2, Mar. 1985, pp. 73-79.

[14] D. Ingalls, s. Wallace, Y-Y Chow, F. Ludolph, K. Doyle, "Fabrik - A Visual Programming Environment", *OOPSLA'88 Proceedings*, ACM, pp. 176-190

[15] W. Finzer and L. Gould, "Programming by Rehearsal", *Byte*, Vol. 9, No. 6, June 84, pp. 187-210.

[16] H. Boley, "RELFUN: A Relational/Functional Integration with Valued Clauses", SIGPLAN Notices 21(12), Dec. 1986, pp. 87-98

[17] J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language", *OOPSLA'87 Proceedings*, ACM, pp. 466-481

[18] Walter F. Tichy, Frances J. Newbery, "Knowledge-based Editors for Directed Graphs", In Howard K. Nichols and Dan Simpson, editors, *1st European Software Engineering Conference,* pp. 101-109, Springer, 1987

[19] A. I. Wasserman, P. A. Pircher, R. J. Muller, "An Object-Oriented Structured Design Method", ACM SIGSOFT, *Software Engineering Notes*, Vol. 14, No. 1, 1989, pp. 32-55

[20] A. I. Wasserman, P. A. Pircher, R. J. Muller, "Concepts of Object-Oriented Structured Design", *Proceedings of Tools'89*, Paris, Nov. 1989

[21] M. Heitz, *"HOOD Reference Manual"*, CISI Ingenierie, Midi Pyrénées, Sep. 1989

[22] Grady Booch, "Object Oriented Design with Applications", Benjamin/Cummings, 1991

[23] M. R. Blaha, W. J. Premerlani, and J. E. Rumbaugh, "Relational Database Design using an Object-Oriented Methology", *Communications of the ACM*, Vol. 31, No. 4, Apr. 1988, pp. 414-427

[24] Frances J. Newbery, "An Interface Description Language for Graph Editors", *Proceedings of the IEEE Workshop on Visual Languages*, Pittsburg, PA, October 10-12, 1988