Diplomarbeit

Softwareaktionen Wiederholen und Rückgängigmachen

Undo/Redo-Konzepte für interaktive Anwendungen am Beispiel des objektorientierten Rahmenwerkes JWAM

Björn Ostermann
Gärtnerstraße 105
20253 Hamburg
5osterma@informatik.uni-hamburg.de
Matrikel-Nr.: 4822776

Januar 2003

Erstbetreuung: Prof. Dr.-Ing. Heinz Züllighoven
Zweitbetreuung: Prof. Dr. Winfried Lamersdorf

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Betreuung:

Prof. Dr.-Ing. Heinz Züllighoven (Erstbetreuer)
Prof. Dr. Winfried Lamersdorf (Zweitbetreuer)

Prof. Dr.-Ing. Heinz Züllighoven

Arbeitsbereich Softwaretechnik (SWT)
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Prof. Dr. Winfried Lamersdorf

Arbeitsbereich Verteilte Systeme und Informationssysteme (VSIS)
Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Danksagung

An dieser Stelle möchte ich allen danken, die mir bei der Erstellung dieser Arbeit behilflich waren. Prof. Dr.-Ing. Heinz Züllighoven danke ich für die Erstbetreuung der Arbeit und seine hilfreichen Tips und Denkanstöße. Bei Prof. Dr. Winfried Lamersdorf bedanke ich mich für die Zweitbetreuung.
Für zahlreiche Hinweise und Anregungen zur praktischen Umsetzung der erarbeiteten Ideen bedanke ich mich bei den Betreuern des Integrationsserverprojektes Petra Becker-Pechau, Holger Breitling und Axel Schmolitzky. Für die gute Zusammenarbeit, die vielen hilfreichen Diskussionen und die große Anzahl von Tips in der Phase der praktischen Umsetzung möchte ich mich bei meinen Mitstreitern in diesem Projekt Simon Ditrich, Erdal Özkan, Johanna Özkan, Robert Tunkel, Rolf Tyzuk und Dennis Weise bedanken.
Bei Andreas Havenstein bedanke ich mich für das Korrekturlesen dieser Arbeit.
Besonderer Dank gebührt Martin Lippert, der als Projektbetreuer und zusätzlich auch als Reviewer mit zahlreichen wertvollen Hinweisen zur Entstehung dieser Arbeit beigetragen hat.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik der Universität Hamburg wird seit langem Software nach dem Werkzeug & Materialansatz entwickelt (siehe [Zül98]). Dieser Entwicklungsansatz (im folgenden kurz WAM-Ansatz genannt) hat das Ziel, den Entwurf und die Konstruktion von anwendungsorientierten Softwaresystemen durch einen Rahmen für die Vorgehensweise zu systematisieren und damit die Qualität der entstehenden Systeme zu steigern. Um die Umsetzung der Ideen des WAM-Ansatzes zu erleichtern, ist am Arbeitsbereich das JWAM-Rahmenwerk entstanden (siehe [Jwa02]). Dieses Java-basierte Rahmenwerk bietet ein Grundgerüst, aus dem relativ einfach WAM-konforme Anwendungen entwickelt werden können.
Ein wichtiger Bestandteil von Softwaresystemen, die nach dem WAM-Ansatz entwickelt werden, sind Werkzeuge. Sie ermöglichen dem Benutzer, den Zustand von Materialien zu sondieren oder zu verändern. Die Bearbeitung der Materialien erfolgt dabei meist in kleinen einzelnen Schritten, die der Benutzer interaktiv mit dem Werkzeug durchführt. Bei dieser interaktiven Bearbeitung kann es schnell vorkommen, daß der Benutzer versehentlich einen falschen Arbeitsschritt ausführt. An dieser Stelle wird eine *Undo*-Funktion benötigt. Es handelt sich hierbei um eine Metafunktion, die es erlaubt, einen zuvor ausgeführten Arbeitsschritt möglichst einfach (meist genügt ein Mausklick) rückgängig zu machen und den ursprünglichen Zustand wiederherzustellen. Ist im Werkzeug keine Undo-Funktion vorhanden, sind dafür unter Umständen viele Arbeitsschritte notwendig. Im schlimmsten Fall muß die Bearbeitung des Materials komplett von vorne beginnen oder das Material ist sogar unwiederbringlich zerstört. Da Fehler bei der Benutzung nie hundertprozentig ausgeschlossen werden können, wird durch die Ausstattung einer Anwendung mit einer Undo-Funktion die Gebrauchstauglichkeit erheblich gesteigert. Dies ist auch einer der Gründe dafür, daß mittlerweile nahezu jedes moderne Softwarepaket eine Undo-Unterstützung bietet.
Eine Anwendung mit Undo-Unterstützung erlaubt dem Benutzer eine andere Herangehensweise als eine Anwendung ohne dieses Hilfsmittel. Mit einer Undo-Unterstützung muß sich der Benutzer nicht bei jedem Arbeitsschritt Gedanken darüber machen, daß ein falscher Mausklick die Arbeit von mehreren Stunden zunichte machen kann. Deswegen muß er nicht jeden Arbeitsschritt genau planen. Er kann auch versuchen, das gewünschte Ergebnis durch Ausprobieren verschiedener alternativer Lösungswege zu erreichen.
Die Arbeitsweise, nicht jeden Schritt nach einem im voraus festgelegten Schema auszuführen, sondern situationsabhängig zu entscheiden, wie zu verfahren ist, ähnelt der des WAM-Leitbildes eines Arbeitsplatzes für eigenverantwortliche Expertentätigkeit (vgl. [Zül98]). Ein Experte in einem Fachgebiet erledigt wechselnde qualifizierte Tätigkeiten, die aufgrund ihrer Vielfältigkeit und Komplexität keine starre Planung ermöglichen. Durch seine Erfahrung und Qualifikation hat er eine Vorstellung, wie er diese Tätigkeit erledigen kann. Dabei handelt es sich aber nur um einen Handlungsrahmen – die exakt auszuführenden Arbeitsschritte sind von Aufgabe zu Aufgabe verschieden. Der Experte wählt für jede Situation den richtigen Arbeitsschritt des Werkzeugs, der zur Erledigung der Aufgabe führt, aus. Da er aber ständig neue Aufgaben erledigen muß, kommt es vor, daß auch er auf Anhieb nicht weiß, welcher Arbeitsschritt der optimale ist. Hier ist eine Undo-Funktion ein sehr wertvolles Hilfsmittel, um Arbeitsschritt für Arbeitsschritt zur gewünschten Lösung zu kommen. Weil auch Experten hin und wieder Fehler machen und dafür eine Korrekturmöglichkeit brauchen, paßt eine Undo-Funktion sehr gut zur unterstützenden Sichtweise (vgl. [Gry96]), die die Grundlage für das Leitbild des Arbeitsplatzes für eigenverantwortliche Expertentätigkeit bildet.

Eine Undo-Funktion hilft Benutzern, die mit einer Anwendung nur zum Teil oder gar nicht vertraut sind, den Umgang mit dieser Anwendung zu erlernen. Da jeder Arbeitsschritt zurückgenommen werden kann, ist es gefahrlos möglich, unbekannte oder neue Funktionen einer Anwendung auszutesten (vgl. [Ols98]). Anwender können sich während ihrer normalen Arbeitstätigkeit mit einer neuen Anwendung vertraut machen oder neue Funktionen in den eigenen Arbeitsablauf integrieren, ohne in Gefahr zu laufen, ihre Arbeitsergebnisse zu zerstören. Eine Undo-Funktion ist somit ein wichtiges Hilfsmittel für die eigenständige Weiterbildung.
Undo-Funktionen sollten sich nicht auf den letzten Arbeitsschritt beschränken. Auch davorliegende Arbeitsschritte sollten rückgängig machbar sein, um weiter in der Vergangenheit liegende Fehler noch korrigieren zu können (vgl. [Mey97]). Eine *Redo*-Funktion sorgt dafür, daß fälschlich zurückgenommene Arbeitsschritte wiederhergestellt werden können.

1.2 Zielsetzung

Das JWAM-Rahmenwerk bietet bisher keine Hilfestellung bei der Entwicklung von Undo-Funktionen für Werkzeuge. Diese Funktionen müssen für jede Anwendung komplett neu entwickelt werden. Da dies sehr aufwendig ist, verzichten bis jetzt trotz der Vorteile für den Anwender alle JWAM-Anwendungen auf eine Undo-Unterstützung.
Das Ziel dieser Diplomarbeit ist es, eine allgemeine Architektur zu entwickeln, mit der Anwendungen mit relativ geringem Aufwand um eine Undo- und Redo-Funktion erweitert werden können. Diese Architektur soll darauf überprüft werden, ob sie sich in die verschiedenen bestehenden Konzepte der JWAM-Werkzeugkonstruktion integrieren läßt und welche Anpassungen dafür notwendig sind. Außerdem soll die entwickelte Architektur daraufhin untersucht werden, ob und mit welchem Aufwand es mit ihr möglich ist, bestehende Anwendungen nachträglich um eine Undo-Funktion zu erweitern oder ob sie nur für Anwendungen geeignet ist, die komplett neu entwickelt werden.

1.3 Vorgehensweise

Auf der Grundlage allgemeiner Betrachtungen über Undo und Redo sowie verschiedener praktischer Beispiele habe ich zwei sich ähnelnde Konzepte zur Realisierung von Undo/Redo-Funktionen für interaktive Anwendungen entwickelt. Die Konzepte unterscheiden sich in ihrer Komplexität und ihren Einsatzvoraussetzungen: Das erste Konzept ist relativ einfach aufgebaut, dafür aber auf eine bestimmte Klasse von Werkzeugen beschränkt. Das zweite Konzept ist universell verwendbar, erfordert dafür aber mehr Aufwand bei seiner praktischen Umsetzung. Für das erste Konzept habe ich verschiedene Implementierungsvarianten der Undo/Redo-Funktion, die prinzipiell auch auf das zweite Konzept übertragen werden können, auf ihre Vor- und Nachteile untersucht. Als Beispielanwendung diente dabei eine kleine graphische Anwendung, das „Graphical Modelling Tool“. Unter Berücksichtigung der erzielten Ergebnisse habe ich anschließend das zweite Konzept implementiert. Hierfür habe ich den Integrationsserver als Beispielanwendung verwendet.
Es handelt sich hierbei um ein Werkzeug, mit dessen Hilfe Entwicklerteams die gemeinsame Codebasis eines Projekts aktuell und fehlerfrei halten können. Seine erste Version ist von Robert Tunkel im Rahmen seiner Studienarbeit entwickelt worden (siehe [Tun01]). Parallel zu dieser Diplomarbeit hat in einem nach den Ideen des Extreme Programmings (siehe [Bec00]) geführten Projekt eine Arbeitsgruppe, an der ich beteiligt war, den Integrationsserver

weiterentwickelt. Dabei haben wir die Anwendung, neben vielen anderen Funktionserweiterungen, auch um eine Unterstützung für Undo und Redo ergänzt.

1.4 Übersicht über weitere Arbeiten zum Thema Undo/Redo

Das Thema Rückgängigmachen und Wiederherstellen ist in der Literatur bereits ausführlich unter verschiedenen Gesichtspunkten untersucht worden. Den theoretischen Hintergrund von Undo- und Redo-Funktionen sowie von verschiedenen Undo-Modellen haben Gordon et al., Leeman und Yang untersucht (siehe [Gor85], [Lee86], [Yan88]). Archer et al. entwickelten mit dem Skript-Modell ein allgemeines Modell zur Beschreibung von Undo-Funktionen (siehe [Arc84]), auf dessen Grundlage verschiedene weitere Undo-Modelle wie das US&R-Modell von Vitter (siehe [Vit84]) und das Triadic-Modell von Yang (siehe [Yan88]) basieren. Einen weiterführenden Ansatz beschreibt Berlage mit dem selektiven Undo-Modell (siehe [Ber94]). Die vier genannten Modelle werden im zweiten Kapitel dieser Arbeit genauer beschrieben und miteinander verglichen.
Wang et al. beschreiben ein Modell, in dem mit Hilfe von speziellen Ereignis-Objekten ein allgemeiner Undo-Mechanismus realisiert wird (siehe [Wan91]). Myers et al. stellen dar, wie mit Hilfe wiederverwendbarer Befehls-Objekte solch ein Undo-Mechanismus implementiert werden kann (siehe [Mye96]). Rathke befaßt sich mit den Anforderungen an Benutzungsschnittstellen für Wiederherstellungsfunktionen (siehe [Rat87]). Mit den besonderen Anforderungen an Undo- und Redo-Funktionen von Anwendungen, in denen mehrere Benutzer kooperativ an einer Aufgabe arbeiten, beschäftigen sich Choudhary et al., Prakash et al. und Ressel et al. (siehe [Cho95], [Pra94], [Res99]).

1.5 Übersicht über die Arbeit

Im Kapitel **Undo-Modelle** werden die Grundlagen von Undo-Funktionen erläutert. Dabei werden verschiedene Undo-Modelle vorgestellt und miteinander verglichen. Undo-Modelle bilden die Grundlage jeder Undo-Funktion und legen ihr generelles, von der Architektur und der Implementierung unabhängiges Verhalten fest. Die untersuchten Modelle unterscheiden sich in ihren Fähigkeiten, in der Vergangenheit ausgeführte Arbeitsschritte rückgängig zu machen. Lineare Undo-Modelle erlauben es nur, Arbeitsschritte in exakt umgekehrter Reihenfolge wie bei der Ausführung zurückzunehmen. Nichtlineare Modelle erlauben zusätzlich, bei der Rücknahme bestimmte Arbeitsschritte zu überspringen. Mit selektiven Undo-Modellen ist es möglich, einmal ausgeführte Arbeitsschritte in beliebiger Reihenfolge rückgängig zu machen und wiederherzustellen.
Das Kapitel **Realisierung von Undo** beschäftigt sich mit der konkreten Umsetzung von Undo-Funktionen. Dazu werden zunächst allgemeine Wiederherstellungsstrategien erläutert. Anschließend werden die Architekturen und die Implementierungen von drei unterschiedlich aufgebauten Undo-Funktionen vorgestellt und miteinander verglichen. Es handelt sich dabei um eine allgemeine Undo-Funktion, die als einzige Voraussetzung die Verwendung des Befehlsmusters hat, die im Swing-Toolkit von Sun Microsystems integrierte Undo-Funktion sowie die Undo-Funktion, die Bestandteil des JHotDraw-Rahmenwerks ist.
Es folgt ein Kapitel, in dem **Undo-Konzepte für manipulatorbasierte Werkzeuge** vorgestellt werden. Anhand des Graphical Modelling Tools werden verschiedene Implementierungsvarianten einer Undo-Funktion, die auf diesem Konzept basiert, beschrieben und miteinander verglichen. Die dabei verwendete Beispielanwendung basiert auf einer speziellen Klasse von JWAM-Werkzeugen, den manipulatorbasierten Werkzeugen.

Schließlich folgt im Kapitel **Undo/Redo-Konzepte für Werkzeuge mit IAK/FK-Trennung** die Vorstellung eines unabhängig vom eingesetzten Werkzeugtyp verwendbaren Undo-Konzepts. Zunächst werden der grundlegende Aufbau und anschließend einige Erweiterungen dieses Konzepts vorgestellt. Am Beispiel des Integrationsservers wird gezeigt, wie mit diesem Konzept eine bestehende Anwendung nachträglich um eine Undo-Funktion ergänzt werden kann.
Den Abschluß der Arbeit bildet eine **Zusammenfassung** der erzielten Ergebnisse und ein **Ausblick** auf offene Fragen.

1.6 Graphische und sprachliche Konventionen

Alle in dieser Arbeit aufgeführten Klassen-, Objekt und Zustandsdiagramme entsprechen dem UML-Standard in der Version 1.4 (siehe [Uml01]). Die übrigen Abbildungen werden im Text oder in den Zeichnungen selbst erläutert.
Wichtige Begriffe werden zur Hervorhebung bei ihrer ersten Verwendung in *kursiver* Schrift dargestellt. Programmcode und direkt aus dem Programmcode übernommene Bezeichnungen werden in `Courier`-Schrift geschrieben.
Die Namen aller aus konkreten Programmbeispielen stammenden Klassen und Methoden sind in englischer Sprache verfaßt, um den Bezug zu den Originalprogrammen zu erhalten. Zusätzlich wurden einige Begriffe, für die es keine oder nur eine künstlich wirkende deutsche Entsprechung gibt, nicht übersetzt. Insbesondere „Undo“ und „Redo“ als zentrale Begriffe dieser Arbeit sind hier hervorzuheben. Diese Begriffe lassen sich zwar einigermaßen verständlich mit „Zurücknahme“ und „Wiederherstellung“ übersetzen, spätestens bei zusammengesetzten Begriffen wirkt die deutsche Übersetzung aber konstruiert. Zudem wird diese Übersetzung in deutschsprachigen Programmen nicht einheitlich verwendet. Während in englischsprachigen Programmen die ausschließlich benutzten Bezeichnungen Undo und Redo existieren, variiert die deutsche Übersetzung je nach Anwendung. Daher wurde größtenteils auf eine Übersetzung dieser beiden Begriffe verzichtet.

2 Undo-Modelle

Es gibt drei Gruppen von Unterstützungsfunktionen, die einem Benutzer einer Anwendung helfen, Fehler zu vermeiden oder ungewollte Situationen zu umgehen (vgl. [Yan88]). Die erste dieser Gruppen umfaßt *Flucht*-Funktionen (escapes). Sie ermöglichen dem Benutzer, Situationen zu verlassen, in denen ein Befehl für eine Anwendung zum Teil oder schon vollständig formuliert ist, aber noch nicht ausgeführt wurde (beispielsweise während der Eingabe von Parametern für eine Funktion). Die zweite Gruppe der Unterstützungsfunktionen umfaßt *Abbruch*-Funktionen (stops). Sie dienen dazu, Befehle zu stoppen oder zu unterbrechen, während sie ausgeführt werden. In der dritten Gruppe finden sich *Wiederherstellungs*-Funktionen (undos). Mit ihnen kann der Effekt eines ausgeführten Befehls aufgehoben und der ursprüngliche Zustand einer Anwendung wiederhergestellt werden. Unterstützungsfunktionen dieser Gruppe sind der Schwerpunkt dieser Arbeit.
Für den grundlegenden Funktionsumfang einer Undo-Funktion, der unabhängig von konkreten Anwendungen und Implementierungen ist, wird in der Literatur der Begriff *Undo-Modell* verwendet. Das Undo-Modell legt fest, wie viele und in welcher Reihenfolge Arbeitsschritte rückgängig gemacht werden können und ob es möglich ist, auch zurückgenommene Arbeitsschritte erneut auszuführen (Redo).
Dieses Kapitel beginnt mit einem historischen Überblick über die Entstehung und den Einsatz von Undo-Funktionen. Anschließend folgt ein Unterkapitel, in dem verschiedene allgemeine Grundlagen von Undo-Funktionen beschrieben und erläutert werden. Der restliche Teil dieses Kapitels befaßt sich mit Undo-Modellen. Vorgestellt werden verschiedene einfache, lineare, nichtlineare und selektive Undo-Modelle. Abschließend folgt ein Vergleich dieser Modelle mit dem Ziel, das für die Entwicklung einer Undo-Funktion für JWAM am besten geeignete auszuwählen.

2.1 Historischer Überblick

Funktionen zur Wiederherstellung früherer Zustände wurden in Softwaresystemen zunächst vor allem bei Datenbankprogrammen eingesetzt (vgl. [Arc84]). Diese Funktionen, die sogenannten *Rollback*-Funktionen, dienen hauptsächlich dazu, eine Datenbank in einen konsistenten Zustand zurück zu versetzen, wenn eine Transaktion (eine, oft mehrteilige, Operation auf einer Datenbank, die nur als Ganzes ausgeführt werden darf, vgl. [Sch97]) nicht vollständig ausgeführt werden kann. Unvollständige Transaktionen können aufgrund falscher oder fehlender Daten oder aufgrund von Systemfehlern auftreten. Eine Rollback-Funktion kann in diesem Fall den Zustand vor der jeweiligen Transaktion wiederherstellen. Da solche Ausnahmesituationen relativ selten auftreten, muß die Benutzung der Rollback-Funktion nicht besonders effektiv in bezug auf Ausführungsgeschwindigkeit und Benutzbarkeit sein (vgl. [Arc84]).
Da Computer lange Zeit nur im Stapelverarbeitungsbetrieb (Batch) betrieben wurden, bei dem der Benutzer nach dem Start eines Programms keinen weiteren Einfluß auf die Ausführung hatte, wurden für die Wiederherstellungsfunktionen ähnliche Techniken wie in Datenbanken verwendet: Batchsysteme konnten in den Zustand vor der Ausführung eines Programms zurückversetzt werden, wenn zuvor eine Sicherung der Daten angelegt wurde. Die Erstellung der Datensicherung mußte dabei manuell vom Benutzer veranlaßt werden (vgl. [Thi90]).
Erst mit dem Wechsel zu interaktiven Anwendungen (zunächst ausschließlich textbasierte) entstand der Bedarf, Korrekturen an den zuvor getätigten Eingaben ebenfalls interaktiv durchzuführen. Die ersten Anwendungen, die eine Undo-Unterstützung besaßen, waren hauptsäch-

lich Programme zum Editieren und Formatieren von Texten (vgl. [Lee86]). Die erste Implementierung einer Wiederherstellungsfunktion in einer umfangreicheren Anwendung erfolgte in Interlisp. Es handelt sich hierbei um eine Entwicklungsumgebung für die Programmiersprache Lisp (vgl. [Tei78]).
Bei anderen Anwendungen wurden Undo-Funktionen zum Teil wesentlich später zum Standard. Erst nachdem sich graphische Oberflächen mit einem systemweit einheitlichen Benutzungsmodell durchgesetzt haben, sind zumindest die meisten Standardanwendungen mit einer Undo-Funktion ausgestattet.

2.2 Grundlagen von Undo

Dieses Unterkapitel beschäftigt sich mit verschiedenen grundlegenden Punkten, die bei der Entwicklung einer Undo-Funktion zu beachten sind. Zunächst wird auf die im folgenden häufig benutzten Begriffe Befehl und Befehlsgeschichte eingegangen. Anschließend folgt eine Aufstellung von Anforderungen, die Undo-Funktionen in Abhängigkeit von den zugehörigen Aktionen erfüllen sollten. Es folgt ein Abschnitt, der sich mit der Granularität und dem Materialbezug von Undo-Funktionen befaßt. Abschließend erfolgt eine Klassifizierung der in den folgenden Kapiteln vorgestellten Undo-Modelle.

2.2.1 Befehl und Befehlsgeschichte

Jede Anwendung mit Undo-Unterstützung muß, unabhängig von dem eingesetzten Modell und unabhängig von der Implementierung, Informationen über ausgeführte *Befehle* sichern. Befehle (commands) sind vom Benutzer veranlaßte Aktionen, die jeweils genau einen Arbeitsschritt umfassen. Befehle werden immer als Ganzes ausgeführt und können somit auch nur als Ganzes zurückgenommen werden. Die Informationen über ausgeführte Befehle werden in der *Befehlsgeschichte* (command history) gespeichert. Eine Undo-Funktion kann diese Information benutzen, um ausgeführte Befehle rückgängig zu machen. Der Umfang der Befehlsgeschichte kann je nach Undo-Modell variieren: Im einfachsten Fall umfaßt sie jeweils nur den letzten ausgeführten Befehl, bei komplexeren Undo-Modellen kann sie aus baumartigen Strukturen aller bisher ausgeführten Befehle bestehen. Die maximale Größe der Befehlsgeschichte ist meist begrenzt, um Speicherplatz zu sparen und Speicherüberläufe zu verhindern. Entweder ist die maximale Anzahl der gespeicherten Befehle oder die Größe des höchstens in Anspruch zu nehmenden Speichers festgelegt. Wenn diese Grenze überschritten wird, werden die ältesten Informationen gelöscht.
Mit den Informationen in der Befehlsgeschichte ist, neben der Unterstützung von Undo, auch die Rekonstruktion von Arbeitsvorgängen nach unbeabsichtigten Programmabbrüchen, wie z.B. Systemabstürzen, möglich. Dafür ist es notwendig, daß die Befehlsgeschichte nach jedem Arbeitsschritt auf einem persistenten Speichermedium wie der Festplatte gesichert wird. Nach einem Programmabsturz und einem anschließenden Neustart kann die Anwendung erkennen (z.B. daran, daß die Befehlsgeschichte nicht leer ist), daß sie nicht korrekt beendet wurde. Ein bestehendes Material, das beim Absturz bearbeitet wurde, wird wieder eingelesen und alle Befehle aus der Befehlsgeschichte werden erneut ausgeführt. Diese Technik wird auch von einer sehr einfachen Wiederherstellungsstrategie, der „Strategie des kompletten erneuten Ausführens“ (complete rerun strategy), eingesetzt (siehe Kapitel 3.1).

2.2.2 Anforderungen an Undo-Funktionen

Grundsätzlich sollte es eine Undo-Funktion erlauben, möglichst alle Befehle rückgängig zu machen, die in der zugehörigen Anwendung vorhanden sind. Ist die Undo-Funktion nur auf bestimmte Befehle beschränkt, so täuscht sie dem Benutzer eine nicht vorhandene Sicherheit vor. Es ist sehr frustrierend für den Benutzer, wenn er, vertrauend auf die Undo-Funktion, völlig unerwartet mit einem nichtkorrigierbaren Fehler konfrontiert wird (vgl. [Thi90]).
Es gibt allerdings Aktionen, die sich nicht rückgängig machen lassen (vgl. [Ols98]). Dazu gehören alle Aktionen, die über die Systemgrenzen hinausgehende (Seiten-)Effekte besitzen. Beispielsweise läßt sich das Verschicken einer E-Mail nicht rückgängig machen. Man kann zwar eine zweite E-Mail an den Adressaten hinterherschicken und ihm sagen, er solle die erhaltenen Informationen „vergessen“. Dies wird aber trotzdem nicht denselben Effekt haben, wie eine nie versandte Nachricht. Ein weiterer nicht zu vernachlässigender Seiteneffekt ist die Zeit, die vergeht, wenn eine Aktion ausgeführt wird. Da vergangene Zeit nicht zurückdrehbar ist, kann in Anwendungen, die von der Systemzeit abhängig sind, niemals exakt der Zustand wiederhergestellt werden, der vor der Ausführung eines Befehls vorhanden war (vgl. [Arc84]).
Aktionen, die durch eine vorhandene Undo-Funktion nicht rückgängig gemacht werden können, sollten so gestaltet sein, daß dies dem Benutzer deutlich wird (vgl. [Thi90]). Eventuell ist es notwendig, eine Sicherheitsabfrage (z.B. „Wollen Sie das Programm ohne Sicherung verlassen?“) einzubauen.
Es muß allerdings nicht für jede Aktion einer Anwendung eine Undo-Funktion entwickelt werden. Eine Undo-Funktion ist nicht notwendig für reine sondierende Aktionen, also für Aktionen, die Werte oder Zustände nur auslesen, sie aber nicht verändern (Vgl. [Yan88]). Da diese Aktionen den Anwendungszustand nicht verändern, müssen sie auch keinen eigenen Zustand sichern. Für Hilfsaktionen, die für das eigentliche Programm nur eine unterstützende Funktion haben (z.B. der Aufruf einer Online-Hilfe), ist ebenfalls keine Undo-Funktion notwendig. Diese Aktionen werden nicht in die Befehlsgeschichte eingefügt und sind somit nach der Ausführung für den Benutzer nicht mehr über die Undo- oder Redo-Funktion zugreifbar.
Für Funktionen, die nur die Darstellung an der Oberfläche beeinflussen, wie z.B. Scrollen oder Selektieren von bestimmten Elementen aus einer Liste, gilt in den meisten Anwendungen dasselbe: Sie werden nicht als eigenständige Befehle in der Befehlsgeschichte geführt und sind somit auch nicht zurücknehmbar (vgl. [Ras00]).
Im Gegensatz dazu schlägt Myers vor, auch für diese Aktionen die Undo- und die Redo-Funktion zu implementieren. Dadurch soll es möglich sein, komplexe Selektionen wiederherzustellen und eine Art Lesezeichen auf Elemente einer Liste zu legen, zwischen denen mit Undo und Redo hin- und hergesprungen werden kann (vgl. [Mye96]). Diese Erweiterung der normalen Handhabung von Undo ist allerdings wenig verbreitet und meiner Meinung nach auch nicht notwendig, da nicht die Selektion, sondern die Aktion, die mit den selektierten Elementen durchgeführt wird, relevant für die Undo-Funktion ist.
Für bestimmte Anwendungen ist eine Undo-Funktion gar nicht gewollt. Hierzu zählen Spiele, wie z.B. Computer-Schach. Hier ist das Zurücknehmen von Zügen mit (Selbst-)Betrug gleichzusetzen, solange man sich nicht in einem Lernmodus befindet. Ähnliches gilt auch bei Trainingsprogrammen für Notfallsituationen, bei denen der Benutzer lernen muß, keine Fehler zu begehen (vgl. [Thi90]).

2.2.3 Granularität und Materialbezug von Undo-Funktionen

Bei der Entwicklung von Undo-Funktionen muß man sich auch Gedanken über die Granularität und den Bezug der Undo-Funktion machen (vgl. [Abo91]). Die Granularität legt fest,

was als ein zurücknehmbarer Arbeitsschritt gilt. Zu kleine Schritte (z.B. bei einer Texteingabe jeder einzelne Buchstabe) sind für den Benutzer unpraktisch, da er dann zu oft die Undo-Funktion benutzen muß, um einen Arbeitsvorgang rückgängig zu machen und verbrauchen zudem wegen der großen Anzahl zu sichernder Schritte sehr viele Systemressourcen. Bei zu großen Schritten besteht die Gefahr, daß nach einem Fehler Arbeit verloren geht. Hier gilt es für den Entwickler, einen Kompromiß zu finden.
Bei Anwendungen, in denen mehrere Materialien gleichzeitig bearbeitet werden können (z.B. bei einem Texteditor mit mehreren Fenstern), kann sich die Undo-Funktion sowohl global auf alle Materialien, als auch lokal auf jeweils ein einziges Material beziehen. Eine lokale Undo-Funktion macht nur die Arbeitsschritte rückgängig, die im gerade aktuellen Material ausgeführt wurden. Für jedes Material wird hierbei eine eigene Befehlsgeschichte angelegt. Eine globale Undo-Funktion benutzt nur eine Befehlsgeschichte für alle Materialien. Mit der Undo-Funktion wechselt der Fokus auf das Material, in dem der Befehl, der zurückgenommen werden soll, ursprünglich ausgeführt wurde.
Eine sehr interessante Lösung benutzt die Entwicklungsumgebung IDEA der Firma IntelliJ (siehe [Int02]): Sie besitzt zwei Undo/Redo-Funktionen. Die erste Undo/Redo-Funktion bezieht sich jeweils lokal auf den gerade bearbeiteten Quelltext, mit der zweiten kann zwischen verschiedenen Quelltexten in der Reihenfolge, wie sie bearbeitet wurden, hin- und hergesprungen werden.

2.2.4 Klassifizierung von Undo-Modellen

Die in den folgenden Kapiteln vorgestellten Undo-Modelle lassen sich nach Yang in zwei Klassen einteilen, *primitive Undo-Modelle* und *Meta-Undo-Modelle* (vgl. [Yan88]). Bei den primitiven Undo-Modellen verhält sich ein Undo-Befehl wie jeder andere normale Befehl. Ein ausgeführter Undo-Befehl wird in der Befehlsgeschichte geführt und kann durch einen weiteren Undo-Befehl rückgängig gemacht werden. Zu den primitiven Undo-Modellen gehören das Single-Undo-Modell und das History-Undo-Modell. Alle übrigen vorgestellten Undo-Modelle gehören zu den Meta-Undo-Modellen. Sie zeichnen sich dadurch aus, daß bei ihnen die auf der Befehlsgeschichte arbeitenden Befehle (Undo, Redo, bei einigen Modellen weitere) nicht wie normale Befehle den Zustand der Anwendung verändern, sondern gewissermaßen auf diesen normalen Befehlen arbeiten. Daher werden sie als Meta-Befehle bezeichnet (vgl. [Dix97]). Meta-Befehle werden nicht mit in die Befehlsgeschichte aufgenommen. Deshalb ist später auch nicht nachvollziehbar, wann sie benutzt wurden.
Eine weitere Möglichkeit zur Klassifizierung von Undo-Modellen besteht in der Eigenschaft, Befehle bei einer erneuten Ausführung (Redo) nie in einem anderen als dem ursprünglichen Zustand auszuführen und Befehle nur aus dem Zustand zurückzunehmen, den sie nach ihrer Ausführung inne hatten. Diese Eigenschaft wird nach Berlage als *stabile Ausführungseigenschaft* (stable execution property) bezeichnet. Er definiert sie folgendermaßen:

> *„A command is always redone in the same state that it was originally executed in, and is always undone in the state that was reached after the original execution.“* ([Ber94], S. 274)

Diese Eigenschaft hat erhebliche Vorteile für die Entwicklung einer Undo-Funktion: Es müssen keine Abhängigkeiten zwischen einzelnen Befehlen berücksichtigt werden, die unter Umständen verletzt werden könnten. Es reicht aus, den vollständige Zustand der Anwendung wiederherstellen zu können. Diese Eigenschaft erfüllen mit Ausnahme des uneingeschränkten linearen Undo-Modells alle einfachen und linearen Undo-Modelle.

2.3 Einfache Undo-Modelle

Für den Benutzer einer interaktiven Anwendung ist jede Form von Unterstützung bei der Korrektur von Fehlern hilfreich. Als einfachstes Hilfsmittel hierzu sind die auf jeder Tastatur vorhandenen „Zurücksetzen“- und „Entfernen“-Tasten zu nennen. Sie erlauben es, den jeweils letzten eingegebenen Buchstaben zu löschen und stellen somit eine Art Undo-Funktion für Texteingaben dar (vgl. [Thi90]).
Dieser sehr einfache Korrekturmechanismus ist aber keine Undo-Funktion im eigentlichen Sinne. Änderungen, die nicht allein auf der Eingabe von Text basieren (z.B. Löschen eines Absatzes, Änderung der Schriftgröße oder Funktionen, die gar nichts mit Textbearbeitung zu tun haben), erfordern weiterführende Ansätze. Erforderlich ist eine Undo-Funktion, die für alle Befehle verfügbar ist und somit Änderungen jeder Art rückgängig machen kann. Für diese Undo-Funktion muß zunächst ein passendes Benutzungsmodell, das Undo-Modell, festgelegt werden.

2.3.1 Single-Undo-Modell

Ein einfaches Undo-Modell ist das *Single-Undo-Modell* (vgl. [Man96]). In diesem Modell gibt es nur einen Undo-Befehl, der auch auf sich selbst angewandt werden kann. War die letzte ausgeführte Aktion ein normaler Bearbeitungsschritt, dann wird durch die Ausführung des Undo-Befehls der vorige Zustand wiederhergestellt. Wenn die zuletzt ausgeführte Aktion ein Undo-Befehl war, dann wird der Zustand vor dem ersten Undo wiederhergestellt. Hier wird aus der Undo- eine Redo-Funktion. Da nur jeweils zwischen den beiden letzten Zuständen hin- und hergewechselt werden kann, wird diese Form des Undos auch *Flip-Undo* genannt (vgl. [Thi90]).
Dieses Undo-Modell war lange die Basis für die Undo-Funktion in der weitverbreiteten Apple Macintosh Klassenbibliothek MacApp (siehe [Sch86]). Daher war dieses Undo-Modell lange Zeit vorherrschend auf dem Apple Macintosh. Auf diesem Undo-Modell basierende Undo-Funktionen fanden sich aber auch in den älteren Versionen von Mircosoft Word für das Betriebssystem MS-DOS.
Für den Entwickler hat dieses Undo-Modell den Vorteil, daß nur der aktuelle und der jeweils vorausgehende Zustand der Anwendung wiederhergestellt werden muß. Daher muß keine vollständige Befehlsgeschichte verwaltet werden und der maximal benötigte Speicherplatz ist gering. Dieses Modell ist somit relativ einfach zu implementieren.
Aus Benutzersicht ist die Beschränkung auf einen Undo-Schritt allerdings für die meisten Anwendungen nicht ausreichend. Fehler werden oft erst nach weiteren Arbeitsschritten erkannt und sind somit von einer Undo-Funktion mit diesem Undo-Modell nicht mehr rückgängig zu machen. Daher ist eine auf dem Single-Undo-Modell basierende Undo-Funktion nur für sehr kleine Anwendungen interessant, bei denen sich der Aufwand für die Entwicklung einer umfangreicheren Undo-Funktion nicht lohnt. Dieses Modell scheidet deswegen als Grundlage für die zu implementierende allgemeine Undo-Funktion aus.

2.3.2 History-Undo-Modell

Das *History-Undo-Modell* (Vgl. [Pra94]) ist eine Erweiterung des im vorigen Abschnitt beschriebenen Single-Undo-Modells. Auch in diesem Modell wird nur ein einziger Undo-Befehl verwendet, der sowohl auf allen normalen Befehlen, als auch auf sich selbst arbeitet. Im Gegensatz zum Single-Undo-Modell ist dieses Modell aber nicht auf einen Undo-Schritt begrenzt. Arbeitsschritte können bis zu einer durch die Anwendung festgelegten Anzahl rückgängig gemacht werden. Die Undo-Befehle werden wie normale Befehle in der Befehlsge-

schichte gespeichert und können später zurückgenommen werden. Auf diese Art wird in dem History-Undo-Modell eine Redo-Funktion realisiert.
Die Benutzung dieses Undo-Modells ist sehr gewöhnungsbedürftig und führt leicht zu Fehlbedienungen. Die Funktionsweise des History-Undo-Models sieht folgendermaßen aus:
Durch wiederholtes Ausführen der Undo-Funktion können zuvor ausgeführte Befehle rückgängig gemacht werden. Sobald ein anderer als der Undo-Befehl ausgeführt wird, springt der *Undo-Zeiger* (ein Zeiger, der auf den zuletzt zurückgenommenen Befehl deutet) an das Ende der Befehlsgeschichte. Hier wird der neue Befehl angehängt. Wird jetzt erneut Undo aufgerufen, startet die Undo-Funktion wieder vom Ende der Liste. Gespeicherte Undo-Befehle werden jetzt genauso wie normale Befehle behandelt und entsprechend von dem Undo-Befehl zurückgenommen. Mit dem folgenden Beispiel soll die Funktionsweise des History-Undo-Modells veranschaulicht werden. (In Abbildung 2.1 stehen a bis e für normale Befehle, U steht für den Undo-Befehl, | ist die Position des Undo-Zeigers.)

	ausgeführte Befehle	Befehls-geschichte	wirksame Befehle	Anmerkung
1.	abc	abc|	abc	
2.	abcUU	a|bcUU	a	2x Undo hebt c + b auf
3.	abcUUd	abcUUd|	ad	neuer Befehl: Undo-Zeiger zurück ans Ende
4.	abcUUdUU	abcU|UdUU	ab	1.Undo hebt d auf, 2.Undo hebt „Undo b“auf
5.	abcUUdUUe	abcUUdUUe|	abe	neuer Befehl: Undo-Zeiger zurück ans Ende
6.	abcUUdUUe UUUUUU	abc|UUdUUe UUUUUU	abc	zurück zu 1.

Abbildung 2.1: Beispiel History-Undo-Modell

An diesem Beispiel wird deutlich, daß es mit zunehmender Länge der Befehlsgeschichte und der Anzahl zwischenzeitlich ausgeführter Undo-Befehle immer schwieriger wird, einen bestimmten Zustand in der Vergangenheit zu erreichen. Ein Vorteil des History-Undo-Modells gegenüber dem im nächsten Kapitel vorgestellten beschränkten linearen Undo-Modell ist, daß jeder Zustand, den eine Anwendung in der Vergangenheit inne hatte, wiederhergestellt werden kann. Ob zwischendurch die Undo-Funktion benutzt wurde, spielt nur in sofern eine Rolle, als daß dadurch die Anzahl der nötigen Undo-Schritte unter Umständen stark ansteigt.
Das History-Undo-Modell wird von den Unix-Texteditoren „vi“ und „Emacs“ in ihren Undo-Funktionen benutzt, ist sonst aber nicht sehr verbreitet. (Die Verwendung der Emacs-Undo-Funktion wird in [Ema02] ausführlich beschrieben.)
Aufgrund seines für Benutzer nicht sehr intuitiven Benutzungsmodells erscheint dieses Undo-Modell nicht optimal geeignet für die zu entwickelnde allgemeine Undo-Unterstützung zu sein.

2.4 Lineare Undo-Modelle

Lineare Undo-Modelle sind dadurch charakterisiert, daß man sich bei ihnen mit Hilfe von Meta-Befehlen linear vorwärts (Redo) oder rückwärts (Undo) durch die Befehlsgeschichte bewegen kann (vgl. [Ber93]). Es ist dabei nicht möglich, Befehle zu überspringen oder auf Befehle zuzugreifen, die nicht an der gerade aktuellen Stelle stehen. Verzweigungen in verschiedene Äste der Befehlsgeschichte sind dagegen möglich. Die Befehlsgeschichte besteht, im Gegensatz zum History-Undo-Modell, nur aus den ausgeführten Befehlen. Meta-Befehle werden nicht abgespeichert und werden daher von weiteren Undo- oder Redo-Schritten nicht berücksichtigt. Da bei den im folgenden beschriebenen linearen (und auch bei den nichtlinea-

ren und selektiven) Undo-Modellen die Undo-Funktion nicht auf sich selbst anwendbar ist, wird stattdessen eine Redo-Funktion verwendet, um einen Undo-Befehl rückgängig zu machen.

2.4.1 Eingeschränktes lineares Undo-Modell

Ein einfach zu handhabendes und zugleich sehr weit verbreitetes Undo-Modell ist das *eingeschränkte lineare Undo-Modell* (restricted linear undo, vgl. [Ber94]). Mit diesem Modell können Befehle, wie bereits im vorigen Abschnitt beschrieben, nur in der gleichen Reihenfolge wie bei ihrer ersten Ausführung rückgängig gemacht und erneut ausgeführt werden. Es ist somit möglich, mit Hilfe der Undo- und Redo-Funktion durch die Befehlsgeschichte zu traversieren.
Wird ein neuer Befehl ausgeführt, nachdem zuvor ein oder mehrere andere Befehle mittels Undo rückgängig gemacht wurden, dann wird dieser neue Befehl an der aktuellen Position in die Befehlsgeschichte eingefügt. Alle zuvor durch die Undo-Funktion zurückgenommenen Befehle werden gelöscht. Damit können diese nicht mehr erneut ausgeführt werden. In diesem Punkt unterscheidet sich das eingeschränkte Undo-Modell von dem in Kapitel 2.4.3 vorgestellten uneingeschränkten linearen Undo-Modell. Ein Beispiel für seine Benutzung zeigt die folgende Tabelle. (Die Bezeichnungen sind identisch zu Abbildung 2.1; R steht für den Redo-Befehl.)

	ausgeführte Befehle	Befehls-geschichte	wirksame Befehle	Anmerkung
1.	abc	abc|	abc	
2.	abcUU	a|bc	a	2x Undo hebt c + b auf
3.	abcUUR	ab|c	ab	b durch Redo wiederhergestellt
4.	abcUURd	abd|	abd	neuer Befehl d macht Redo von c unmöglich

Abbildung 2.2: Beispiel eingeschränktes lineares Undo-Modell

Ein Vorteil dieses Undo-Modells ist, daß es für den Benutzer wesentlich offensichtlicher als beim History-Undo-Modell ist, was das Ergebnis eines Undo- oder Redo-Befehls sein wird. Wenn der Benutzer zu einem früheren Zustand der Anwendung zurückkehren möchte, wird er nicht durch zuvor ausgeführte Undo-Befehle irritiert, die ihn zwischen früheren Zuständen hin- und herspringen lassen. Er muß nur die Schritte rückgängig machen, die zum aktuellen Zeitpunkt noch Auswirkungen haben – also nur die, die in der Befehlsgeschichte vor dem aktuellen Befehl stehen.
Ein weiterer Vorteil liegt in der einfachen Implementierbarkeit dieses Modells. Als Datenstruktur für die ausgeführten Befehle wird nur eine einfache Liste benötigt. Auf dieser Liste wird ein Zeiger auf den jeweils zuletzt ausgeführten Befehl bei der Undo-Funktion rückwärts und bei der Redo-Funktion vorwärts bewegt (siehe Abbildung 2.3).

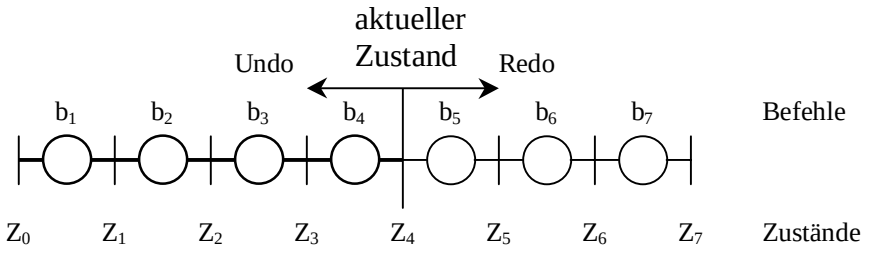

Abbildung 2.3: Befehlsgeschichte des eingeschränkten linearen Undo-Modells

Alternativ kann dieses Modell auch mit zwei Listen, einer Undo- und einer Redo-Liste, implementiert werden. Ausgeführte Befehle werden zunächst in der Undo-Liste gespeichert. Ruft der Benutzer die Undo-Funktion auf, wird der letzte Befehl rückgängig gemacht und in die Redo-Liste verschoben. Mit der Redo-Funktion gelangt er zurück in die Undo-Liste. Nach der Ausführung eines neuen Befehls wird die Redo-Liste geleert.
Dieses Modell ist das bei modernen Anwendungsprogrammen am weitesten verbreitete Undo-Modell. Die meisten Anwendungen, die für das Betriebssystem Microsoft Windows entwikkelt worden sind, aber mittlerweile auch viele Programme für X-Windows oder MacOS, benutzen dieses Undo-Modell.

2.4.2 Lineares Undo-Modell mit Befehlsbaum

Das *lineare Undo-Modell mit Befehlsbaum* (history tree, vgl. [Ber94]) ist eine Erweiterung des eingeschränkten linearen Undo-Modells. Für die Speicherung der Befehlsgeschichte wird statt einer linearen Liste, wie beim eingeschränkten linearen Undo-Modell, ein Baum verwendet. Ein neuer Befehl führt nicht dazu, daß jeweils der Teil der Befehlsgeschichte gelöscht wird, der nach dem aktuellen Zustand folgt. Stattdessen wird ein neuer Zweig in dem Befehlsbaum erzeugt. In Abbildung 2.4 ist ein Befehlsbaum mit einem aktuellen und einem alten Zweig dargestellt.

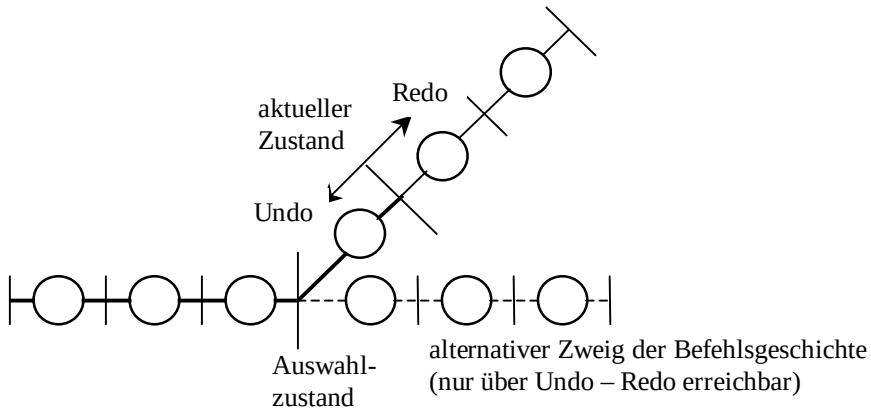

Abbildung 2.4: Befehlsgeschichte des linearen Undo-Modells mit Befehlsbaum

Durch die Undo-Funktion kann weiter wie bisher zu einem vorigen Zustand zurückgekehrt werden. Bei der Redo-Funktion muß der Benutzer entscheiden, welchen Zweig der Befehlsgeschichte er einschlagen möchte, wenn es mehrere Möglichkeiten gibt. Dies kann z.B. durch ein Popup-Menü erfolgen, in dem die zur Auswahl stehenden nachfolgenden Befehle aufgelistet sind. Da der Benutzer aber oft nicht weiß, an welcher Position des Baums der Befehl liegt, bis zu dem er die Befehlsgeschichte wiederherstellen will, kann es hilfreich sein, ihm gleich den kompletten Befehlsbaum als Hilfsmittel zu präsentieren. Abbildung 2.5 auf der folgenden Seite zeigt ein Beispiel, in dem die Benutzung des linearen Undo-Modells mit Befehlsbaum demonstriert wird. R* steht dabei für die Stelle, an der der Benutzer entscheiden muß, welchen Zweig er wiederherstellen möchte.
Die erweiterten Möglichkeiten dieses Undo-Modells erkauft man sich durch eine komplexere Benutzungsschnittstelle als beim beschränkten linearen Undo-Modell. Der Benutzer erreicht nicht mehr automatisch den aktuellsten Zustand der Anwendung, indem er wiederholt die Redo-Funktion benutzt. Er muß bei jeder Verzweigung des Befehlsbaums eine Entscheidung treffen, welcher Zweig der Befehlsgeschichte der richtige ist. Ohne weitere Unterstützung (z.B. Übersicht über alle Befehle im jeweiligen Zweig) kann es sehr kompliziert sein, einen bestimmten Zustand wiederzufinden. Für den Entwickler bedeutet dies erheblichen Mehrauf-

wand, da neben der eigentlichen Undo-Funktion auch noch die Unterstützung für den Benutzer, zum Beispiel die Anzeige des kompletten Befehlsbaums, zu entwickeln ist.

	ausgeführte Befehle	Befehls-geschichte	wirksame Befehle	Anmerkung
1.	abc	abc|	abc	
2.	abcUU	a|bc	a	2x Undo hebt c + b auf
3.	abcUUd	ad| └bc	ad	mit dem Befehl d wird ein neuer Zweig erzeugt, b+c bilden den 2. Zweig
4.	abcUUdUR*	ab|c └d	ab	Undo hebt d auf, bei dem folgenden Redo ist eine Entscheidung für einen Zweig notwendig (hier b)

Abbildung 2.5: Beispiel lineares Undo-Modell mit Befehlsbaum

Der Vorteil des baumbasierten linearen Undos liegt darin, daß man, wie beim History-Undo-Modell, alle in der Vergangenheit liegenden Zustände wieder erreichen kann und somit kein Befehl gelöscht wird.

2.4.3 Uneingeschränktes lineares Undo-Modell

Das *uneingeschränkte lineare Undo-Modell* (eigentlich nur lineares Undo-Modell, hier zur Unterscheidung uneingeschränkt genannt, vgl. [Vit84]) ähnelt dem in Kapitel 2.4.1 beschriebenen beschränkten linearen Undo-Modell. Es unterscheidet sich nur in dem Verhalten, das auftritt, wenn Befehle mit der Undo-Funktion zurückgenommen werden und anschließend ein neuer Befehl ausgeführt wird: Beim uneingeschränkten linearen Undo-Modell wird die Redo-Liste nicht gelöscht, sie bleibt unverändert. Die Befehle aus der Redo-Liste können anschließend mit der Redo-Funktion wiederhergestellt werden. Damit erlaubt es dieses Modell, Befehle an einer beliebigen Stelle der Befehlsgeschichte einzufügen. Es ist aber auch mit diesem Modell nicht möglich, einen beliebigen Befehl rückgängig zu machen. Dies ist nur mit nichtlinearen und selektiven Undo-Modellen möglich, die in den nächsten Kapiteln vorgestellt werden. Auch hierzu folgt ein Beispiel:

	ausgeführte Befehle	Befehls-geschichte	wirksame Befehle	Anmerkung
1.	abc	abc|	abc	
2.	abcUU	a|bc	a	2x Undo hebt c + b auf
3.	abcUUd	ad|	ad	Befehl d wird eingefügt
4.	abcUUdRR	adbc|	adbc	durch 2x Redo werden bc nach d wiederhergestellt

Abbildung 2.6: Beispiel uneingeschränkte lineares Undo-Modell

Diese gegenüber den bisher vorgestellten Undo-Modellen erweiterte Funktionalität bringt aber auch Nachteile mit sich: Der eingefügte Befehl kann unter Umständen Auswirkungen auf die nachfolgenden Befehle haben, oder er kann an der Stelle, an der eingefügt wurde, nicht mehr ausgeführt werden – die stabile Ausführungseigenschaft wird somit nicht mehr erfüllt. Für dieses Undo-Modell ist daher eine Prüfung der Abhängigkeiten zwischen den einzelnen Befehlen notwendig.

2.5 Nichtlineare Undo-Modelle

Die bisher vorgestellten einfachen und linearen Undo-Modelle (mit Ausnahme des uneingeschränkten linearen Undo-Modells) orientieren sich zum einen an den Anforderungen und Vorgaben der Entwickler. Sie zielen auf die sichere und zugleich einfache Aufrechterhaltung des konsistenten Zustands ihres Modells. Zum anderen haben sie, wenn man einmal von dem History-Undo-Modell absieht, den Vorteil, für den Anwender leicht verständlich und damit einfach zu benutzen zu sein.
Gleichzeitig schränken diese Modelle allerdings die Undo-Funktionalität erheblich ein. Bei der Arbeit mit einem Anwendungsprogramm werden Fehler oft nicht sofort, sondern erst später erkannt. Dann hat der Anwender aber meist schon weitere Arbeitsschritte ausgeführt. Mit linearen Undo-Modellen ist es nur möglich, den in der Vergangenheit liegenden Fehler zu korrigieren, wenn alle danach durchgeführten Arbeitsschritte ebenfalls zurückgenommen werden. Diese Befehle nach der Korrektur wiederherzustellen ist entweder gar nicht möglich (beschränktes lineares Undo-Modell) oder nur dann, wenn es sich bei der Korrektur um das Einfügen eines neuen Befehls handelt (uneingeschränktes lineares Undo-Modell). Je mehr Arbeitsschritte nach fehlerhaften Befehlen ausgeführt wurden, desto weniger lohnt es sich für den Anwender, die Undo-Funktion zu benutzen. Der Aufwand für eine Korrektur von Hand ist oft geringer, als für ein Korrektur mit der Undo-Funktion.
Diesen Nachteil versuchen die nichtlinearen Undo-Modelle zu überwinden. Sie erlauben es, nahezu beliebige Änderungen an der Befehlsgeschichte vorzunehmen und somit fast jede Art von Fehler nachträglich zu korrigieren.
In diesem Unterkapitel werden drei nichtlineare Undo-Modelle vorgestellt. Es handelt sich hierbei um das Skript-Modell, das US&R-Modell und das Triadic-Modell. Das Skript-Modell ist gegenüber den anderen beiden Modell ein Sonderfall, da es sich eigentlich eher um ein umfassendes Konzept handelt, auf dem verschiedene Undo-Modelle aufbauen.

2.5.1 Skript-Modell

Archer, Conway und Schneider stellen mit dem *Skript-Modell* ein weitreichendes Konzept zum Aufbau und zur Benutzung von Undo vor (vgl. [Arc84]). Anhand dieses Konzepts beschreiben sie verschiedene, in ihrer Funktion unterschiedlich umfangreiche Undo-Modelle, die zum Teil den bereits vorgestellten Modellen ähneln. Eines dieser Modelle, von Archer et al. *Truncate/reappend* genannt, erlaubt es, mit Hilfe von Metabefehlen die Befehlsgeschichte beliebig umzuordnen und ermöglicht somit weitreichendere Änderungen als die bisher vorgestellten einfachen und linearen Undo-Modelle. Die in den folgenden beiden Abschnitten beschriebenen Modelle (US&R-, Triadic-Modell) bauen auf den Ideen des Skript-Modells auf.
Mit diesem Modell entwickelten Archer et al. einen der ersten universellen Ansätze zur Realisierung einer Undo-Funktion. Als das Skript-Modell entstand war es eher unüblich, daß Anwendungen mit einer Undo-Funktion ausgestattet waren. Zudem ist dieses Modell noch stark an kommandozeilenbasierten Anwendungen orientiert.
Archer et al. betrachten die Interaktion des Benutzers mit dem System als einen aus zwei Schritten bestehenden Vorgang: Ein neuer Befehl wird zuerst vom Benutzer erstellt und anschließend an das System zur Ausführung übergeben. Dabei muß aber nicht zwangsläufig jeder erzeugte Befehl sofort ausgeführt werden. Es können alternativ auch erst mehrere Befehle erstellt werden, die anschließend zusammen zur Ausführung kommen. Ein Extremfall stellt dabei das klassische Batchsystem dar, bei dem der Benutzer die komplette Bearbeitung im voraus plant und nach dem Start des Prozesses keinen Einfluß mehr auf den Ablauf hat. Der andere Extremfall entspricht einem interaktiven System, wie man es heutzutage bei nahezu allen Anwendungen mir graphischer Benutzungsoberfläche vorfindet: Nach jedem ausgeführ-

ten Befehl liefert das System eine visuelle Rückmeldung über den Bearbeitungsfortschritt, die dem Benutzer bei der Auswahl des nächsten Befehls hilft. Die Folge aller ausgeführten Befehle wird in diesem Modell als *Skript* bezeichnet.
Nach dem Skript-Modell gliedert sich somit die Bearbeitung einer größeren Aufgabe mit einem Programm in zwei sich abwechselnde Phasen: In der Bearbeitungsphase erstellt der Benutzer neue Befehle, in der Ausführungsphase arbeitet das System diese Befehle ab. Die typische Bearbeitung des Skripts besteht daraus, daß der Benutzer einen neuen Befehl an das Skript anhängt und das System diesen ausführt. Je nach Modell sind aber auch andere, weitreichendere Änderungen an dem Skript mit einer Reihe von Metabefehlen möglich. Sie erlauben es, Befehle zu erstellen, zu verändern, zu löschen, an anderer Stelle einzufügen oder die Reihenfolge der Befehle zu ändern. In dem Skript werden nur die Ergebnisse dieser Änderungen gespeichert, nicht aber die Metabefehle selbst. Da das Skript selbst ein Objekt ist, kann man die Metabefehle als Befehle auffassen, die genau dieses Objekt bearbeiten können.
Jedes Skript (S) kann in zwei Abschnitte geteilt werden. Der erste Teil besteht aus den ausgeführten (E), der zweite aus den noch auszuführenden Befehlen (P).

$$S: \underbrace{c_1, c_2, ..., c_{i-1}}_{E}, \underbrace{c_i, c_{i+1}, ..., c_n}_{P}$$

Werden bei der Bearbeitung Teile des Skripts verändert, die bereits ausgeführt wurden, dann muß das System einen früheren Zustand wiederherstellen und ab diesem das veränderte Skript erneut durchlaufen. Das neue Skript (S´) besteht aus unveränderten (U') und modifizierten Befehlen (M´).

$$S´: \underbrace{c_1, c_2, ..., c_{j-1}}_{U´}, \underbrace{c´_j, c´_{j+1}..., c´_{n´}}_{M´}$$

Im folgenden werden verschiedene Varianten von Regelsätzen vorgestellt, mit denen die Befehlsgeschichte modifiziert werden kann. Diese fünf Arten der Skriptmodifikation entsprechen der von Archer et al. gewählten Einteilung.

1. „Incremental modification“:
   - Befehle dürfen nur einzeln und nur am Ende eines Skripts eingefügt werden (append)
   ⇒ entspricht normalen interaktiven Programm ohne Undo

2. „Single-truncate“:
   - ein neuer Befehl kann an das Ende des Skripts angehängt werden (append)
   - der letzte Befehl kann aus dem Skript entfernt werden, wenn zuvor ein Befehl eingefügt wurde (truncate)
   ⇒ entspricht dem Single-Undo-Modell (siehe Kapitel 2.3.1)

3. „Repeated-truncate“:
   - ein neuer Befehl kann an das Ende des Skripts angehängt werden (append)
   - wiederholtes entfernen des letzten Befehls aus dem Skript möglich (truncate*)
   ⇒ sämtliche Änderungen sind möglich, Redo aber nur mit Hilfe von kompletter erneuter Ausführung der Befehle durch den Benutzer

4. „Truncate/reappend“:
   - ein neuer Befehl kann an das Ende des Skripts angehängt werden (append)
   - der letzte Befehl wird aus dem Skript entfernt und in einem Zusätzlichen Skript (R) zwischengespeichert (truncate)

- entfernte Befehle können auch nachdem andere Befehle ausgeführt wurden wieder in das Skript eingefügt werden (reappend)

⇒ jegliche Änderungen am Skript sind, wenn auch umständlich, möglich, ohne daß Befehle komplett neu ausgeführt werden müssen

5. „Unrestricted modification“:
   - neue Befehle können an jeder Stelle in das Skript eingefügt werden
   - bestehende Befehle können gelöscht, modifiziert oder in ihrer Reihenfolge geändert werden
   - das Skript kann wie in einem Texteditor beliebig bearbeitet werden

Wie man an der Liste erkennt, betrachten Archer et al. ihr Skript-Modell als eine allgemeine Grundlage, auf der Wiederherstellungsfunktionen unterschiedlichsten Umfangs basieren können. Die Bandbreite reicht dabei von einem Regelsatz, der überhaupt keine Änderungen an einem bestehenden Skript erlaubt, bis zu einem, bei dem beliebige Änderungen an einem bestehenden Skript vorgenommen werden dürfen. Diese Flexibilität geht aber zur Lasten der Benutzbarkeit: Der Anwender muß hierbei eine textuelle Repräsentation der Befehlsgeschichte bearbeiten. Außerdem können Änderungen im Skript unter Umständen unabsehbare Folgen für darauffolgende Befehle haben. Archer et al. bezeichnen die uneingeschränkte Möglichkeit zu Veränderung des Skripts selbst als „*...too powerful and potentially confusing for general use.*“ ([Arc84], S. 17). Daher haben Archer und Conway für ihre Beispielanwendung „COPE“, die auf dem Skript-Modell basiert, auf das unter 4. genannte Benutzungsmodell (truncate/reappend) zurückgegriffen. Bei COPE handelt es sich um eine integrierte Entwicklungsumgebung für Computerprogramme (siehe [Arc81]).

2.5.2 US&R-Modell

Das *US&R-Modell* (Undo, Skip and Redo) von Vitter ist eine Erweiterung des im vorigen Kapitel vorgestellten allgemeinen Skript-Modells (vgl. [Vit84]). Es wurde mit dem Ziel entwickelt, umfangreiche Undo- und Redo-Funktionen zu bieten und gleichzeitig für den Anwender einfach benutzbar zu sein.
Lineare Undo-Modelle sind in ihren Möglichkeiten auf die Befehlsgeschichte einzuwirken sehr begrenzt. Das zeigt sich insbesondere bei zwei häufig gewünschten Änderungen an der Befehlsgeschichte: Zum einen ist es nicht möglich, Befehle in die Befehlgeschichte einzufügen, ohne sämtliche auf den eingefügten folgende Befehle erneut manuell ausführen zu müssen. Dies Problem tritt nur beim normalen linearen Undo-Modell nicht auf. Zum anderen ist es bei keinem linearen Undo-Modell möglich, Befehle zu entfernen, ohne den folgenden Teil der Befehlsgeschichte zu verlieren. Diese Einschränkungen will Vitter mit dem US&R-Modell überwinden.
Das US&R-Modell benutzt drei Metabefehle – Undo, Skip und Redo –, die auf normalen Befehlen, nicht aber auf sich selbst, arbeiten. Undo hebt den letzten Befehl auf und löscht ihn anschließend aus dem aktuellen Zustand. Den Begriff „Zustand“ definiert Vitter für sein Modell wie folgt:

> *„We define the state of the system to be the sequence of actions still in effect, where an action is either the execution or the skip of a primitive command.“* ([Vit84], S.170)

Skip erlaubt es, Befehle, die zuvor durch Undo zurückgenommen wurden, zu überspringen. Der übersprungene Befehl wird dabei an der aktuellen Stelle in die Befehlsgeschichte eingefügt, aber nicht ausgeführt. Übersprungene Befehle haben zunächst keine Auswirkungen für den Benutzer, können aber den Effekt von späteren Metabefehlen beeinflussen. Mit Redo

werden durch Undo zurückgenommene Befehle erneut ausgeführt und an der aktuellen Position in die Befehlsgeschichte eingefügt. Zurückgenommene Befehle bleiben, im Gegensatz zu den linearen Undo-Modellen, in der Befehlsgeschichte erhalten, wenn ein neuer Befehl ausgeführt wird. Sie können durch die Ausführung von Redo nach dem neuen Befehl wieder in die Befehlsgeschichte eingefügt werden.
Es folgen zwei Beispiele, in denen gezeigt wird, wie im US&R-Modell Änderungen in der Befehlsgeschichte ablaufen.

Beispiel 1: Einfügen von Befehlen

Der Benutzer führt n Befehle A_1, A_2, A_3, ..., A_n aus und stellt dann fest, daß er zwischen den Befehlen A_1 und A_2 die Befehle B_1 und B_2 hätte ausführen müssen. Mit dem US&R-Modell ist dies mit geringem Aufwand möglich: Der Benutzer führt zunächst n-1 mal die Undo-Funktion aus und gelangt so zurück zu dem Zustand, den die Anwendung nach Ausführung von A_1 inne hatte. Anschließend führt er die Befehle B_1 und B_2 aus. Die Befehlsgeschichte besteht jetzt aus den Befehlen A_1 B_1 B_2. Durch (n-1)-malige Benutzung der Redo-Funktion werden die Befehle A_2 bis A_n erneut ausgeführt und wieder an die Befehlsgeschichte angehängt. Die Liste der ausgeführten und für den Endzustand relevanten Befehle besteht jetzt aus A_1 B_1 B_2 A_2 ... A_n.

Beispiel 2: Ersetzen eines Befehls

Angenommen der Benutzer hat n Befehle A_1, A_2, ..., A_n ausgeführt und stellt nun fest, daß er statt des Befehls A_2 besser die Befehle B_1 und B_2 ausführen sollte. Dies kann dadurch erreicht werden, daß als erstes (n-1)-mal Undo ausgeführt wird, um wieder in den Zustand nach Ausführung von A_1 zu gelangen. Ein Aufruf des Skip-Befehls führt nun dazu, daß der Befehl A_2 intern als übersprungen markiert wird. Für den Benutzer hat dies keine äußerlich erkennbaren Auswirkungen. Nun ist es möglich, die neuen Befehle B_1 und B_2 auszuführen und durch (n-1)-malige Benutzung von Redo den Endzustand nach Ausführung von A_n zu erreichen. Allerdings sind die Redo-Schritte nicht mehr eindeutig, da es jeweils möglich ist den übersprungenen Schritt A_2 oder einen der Schritte $A_{x(x:3\leq x\leq n)}$ erneut auszuführen. Daher muß bei jedem Redo die Auswahl getroffen werden, welcher Schritt rückgängig zu machen ist. Abschließend besteht die Befehlsgeschichte aus den folgenden Befehlen:
A_1 ~~A_2~~ B_1 B_2 A_3 ... A_n (Der übersprungene Zustand ist durchgestrichen dargestellt.)
Die aus den beiden Beispielen resultierende Speicherbäume, in dem die Befehlsgeschichten gespeichert sind, sehen in einer vereinfachten Darstellung wie folgt aus:

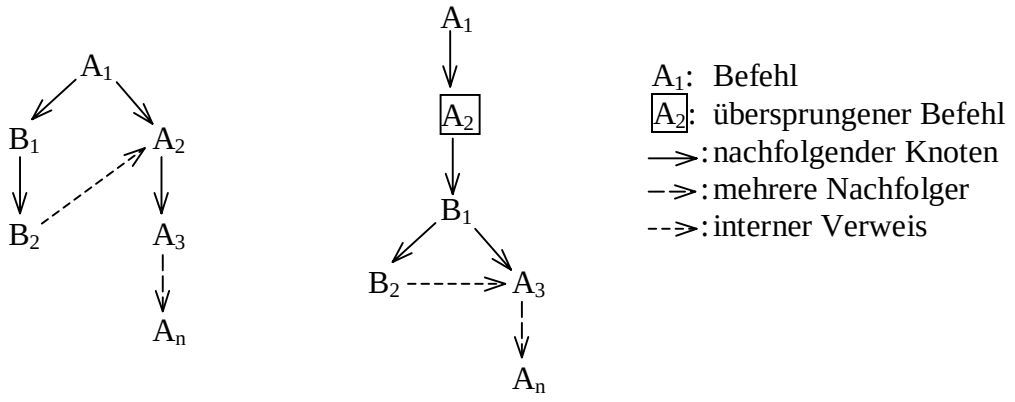

Abbildung 2.7: Speicherbäume zu Beispiel 1 und 2 des US&R-Modells

Mit ähnlichen Kombinationen der Metabefehle Undo, Skip und Redo wie in diesen beiden Beispielen lassen sich nahezu beliebige Änderungen an der Befehlsgeschichte vornehmen. Änderungen in der Reihenfolge der einzelnen Befehle sind genauso denkbar wie selektives Undo (gezieltes Zurücknehmen eines Befehls).
Eines der Probleme des US&R-Modells ist das für den Anwender nicht sehr leicht verständliche Benutzungsmodell. Mit Skip und Redo stehen zwei Befehle zur Verfügung, die es erlauben, durch die Befehlsgeschichte vorwärts zu traversieren. Daher muß sich der Anwender bei jedem Anwendungsschritt, den er wiederherstellen will, überlegen, welcher dieser beiden Befehle der jeweils richtige ist. Zudem besteht bei diesem Modell ein Problem, das bereits beim linearen Undo-Modell mit Befehlsbaum aufgetreten ist: Wenn es eine Auswahlmöglichkeit zwischen mehreren Befehlen gibt, kann der Benutzer nur anhand der in diesem Zustand vorhandenen Informationen (z.B. Namen der Befehle) entscheiden, welchen der Befehle er wiederherstellen will (siehe Kapitel 2.4.2). Er weiß aber nicht, mit welcher der Auswahlmöglichkeiten er zu einem im Baum wesentlich später stehenden Zustand gelangt. Auch hier kann eine Abbildung des kompletten Befehlsbaums als Orientierungshilfe dienen.

Implementierung

Eines der Probleme des US&R-Modells liegt in der aufwendigen Datenstruktur, die zur Speicherung der einzelnen Befehle verwendet wird. In seiner Implementierung benutzt Vitter einen Speicherbaum, der aus bis zu drei verschiedenen Arten von Knoten bestehen kann. In den Knoten sind mit Hilfe von Zeigern Verweise auf die Vater-, Sohn- oder Geschwisterknoten abgespeichert. Zudem gibt es in diesem Speicherbaum einen globalen Zeiger, der auf die aktuelle Position der Befehlsgeschichte deutet. Dies ist der sogenannte Aktuelle-US&R-Zeiger (US&R_current_ptr).
Der Standardknotentyp ist der sogenannte primitive Typ. In ihm werden fünf verschiedene Zeiger abgespeichert. Alle Zeiger, die nicht initialisiert wurden, haben den Wert Null. Die einzelnen Zeiger haben die folgenden Funktionen:

- Befehlszeiger (cmd_ptr): Er zeigt auf die Informationen, die zur Ausführung des Befehls notwendig sind. Unter anderem sind das die Argumente dieses Befehls.
- Do-Undo-Zeiger (do_undo_ptr): Er zeigt auf den Vaterknoten, falls der Befehl dieses Knotens einmal ausgeführt wurde. Wenn dieser Befehl zurückgenommen wird, wird der Aktuelle-US&R-Zeiger auf den Vaterknoten gesetzt. Der Do-Undo-Zeiger bekommt anschließend den Wert „Null“.
- Skip-Undo-Zeiger (skip_undo_ptr): Er zeigt auf den Vaterknoten falls der Befehl dieses Knotens übersprungen wurde. Nach Ausführung von Undo wird ebenfalls der Aktuelle-US&R-Zeiger auf den Vaterknoten gesetzt und anschließend der Skip-Undo-Zeiger auf „Null“ zurückgesetzt.
- Erster-Sohn-Zeiger (first_son): Er zeigt auf den ersten Sohn dieses Knotens. Nach Ausführung von Redo oder Skip wird der Aktuelle-US&R-Zeiger auf diesen Knoten (oder einer seiner Geschwister) gesetzt und der entsprechende Befehl ausgeführt.
- Nächster-Bruder-Zeiger (next_bro): Er zeigt auf den nächsten Bruder dieses Knotens. Jeweils unter allen zusammengehörigen Geschwisterknoten muß der Benutzer einen auswählen, auf dem er Undo bzw. Skip ausführen will.

Die anderen beiden Knotentypen sind der Indirekte-Skip-Knoten und der Indirekte-Do-Knoten. Sie repräsentieren Skip- bzw. Redo-Befehle, die zu einem früheren Zeitpunkt ausgeführt wurden. In Knoten dieser Typen werden nur zwei Zeiger abgespeichert:

- Primitiver-Zeiger: Er zeigt auf den primitiven Knoten, der zu einem früheren Zeitpunkt übersprungen bzw. wiederholt wurde.
- Nächster-Bruder-Zeiger: Er hat hier gleiche Funktion wie im primitiven Knoten.

Mit Hilfe eines Baums aus Knoten der drei genannten Typen wird im US&R-Modell die Befehlsgeschichte gesichert. Solange keiner der Befehle Undo, Skip oder Redo ausgeführt wurde, entspricht der Befehlsbaum einer doppelt verketteten Liste. Die Verkettung besteht aus den Do-Undo-Zeigern und den Erster-Sohn-Zeigern.
Bei der Ausführung von Undo, Skip oder Redo werden je nach Befehl neben dem Aktuellen-US&R-Zeiger auch diverse Zeiger zwischen den einzelnen Knoten verschoben, ergänzt oder entfernt. Auf eine genauere Beschreibung der für die einzelnen Befehle notwenigen Bearbeitungsschritte soll hier aus Platzgründen verzichtet werden. Ausführlich sind diese Schritte von Vitter beschrieben worden (siehe [Vit84]).

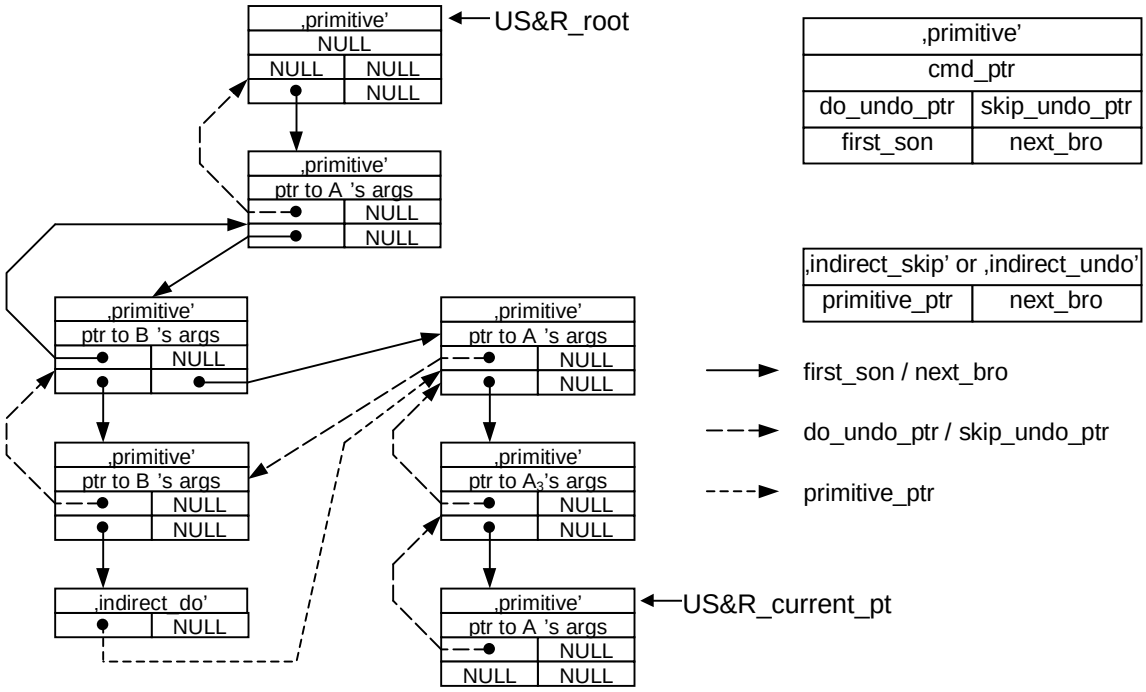

Abbildung 2.8: Interner Aufbau des Befehlsbaums aus Beispiel 1 (aus [Vit84])

In Abbildung 2.8 ist der interne Aufbau des Befehlsbaum aus Beispiel 1 dargestellt. Der Baum besteht aus primitiven Knoten und einem Indirekten-Do-Knoten.

2.5.3 Das Triadic-Modell

Das Triadic-Modell von Yang (vgl. [Yan88]) ist ebenfalls mit dem Ziel entwickelt worden, die durch das lineare Undo-Modell vorgegebenen Grenzen der Wiederherstellungsfunktionen zu überwinden. Gleichzeitig soll auf eine komplizierte Datenstruktur, so wie sie im US&R-Modell verwendet wird, verzichtet werden.
Yangs Modell benutzt, wie das US&R-Modell, drei Metabefehle um die Befehlsgeschichte zu bearbeiten. Bei diesen Befehlen handelt es sich um *undo(n)*, *redo(n)* und *rotate(n)* (Drehen, n steht jeweils für die Anzahl, wie oft der Befehl ausgeführt werden soll). Zusätzlich sorgt *command* dafür, daß ein neuer Befehl ausgeführt und gesichert wird. Das Triadic-Modell benutzt zum Speichern sowohl der ausgeführten wie der zurückgenommenen Befehle je eine Ringliste (eine lineare Liste, bei der das letzte Element wieder auf das erste zeigt). Yang bezeichnet diese Listen als Zustandsliste (state list) und Rücknahmeliste (undone list). Diese

beiden Ringlisten zusammen bilden die Befehlsgeschichte. Die detaillierte Funktionsweise der einzelnen Befehle sieht wie folgt aus:

command: führt einen Befehl aus und fügt ihn am Ende der Zustandsliste ein
undo: entfernt den letzten Befehl aus der Zustandsliste, macht ihn rückgängig und fügt ihn am Ende der Rücknahmeliste ein
redo: entfernt letzten Befehl aus der Rücknahmeliste, führt ihn erneut aus und fügt ihn am Ende der Zustandsliste ein
rotate: dreht die Rücknahmeliste um eine Position; der jeweils am Ende der Liste stehende Befehl kann durch redo erneut ausgeführt werden

Mit diesen vier Befehlen lassen sich, wie beim US&R-Modell, nahezu beliebige Änderungen an der Befehlsgeschichte vornehmen. Die beiden Beispiele aus Kapitel 2.5.2, Einfügen eines Befehls und Ersetzen eines Befehls, lassen sich mit dem Triadic-Modell wie folgt lösen:

Beispiel 1: Einfügen von Befehlen

Hier sieht die Lösung exakt so aus wie beim US&R-Modell. Um nach n ausgeführten Befehlen $A_1...A_n$ die Befehle B_1 und B_2 zwischen A_1 und A_2 einzufügen, sind folgende Schritte notwendig:

1. n-1 mal undo: die letzten n-1 Befehle werden rückgängig gemacht und in der Rücknahmeliste gespeichert
2. command B_1, command B_2: die Befehle B_1 und B_2 werden ausgeführt
3. n-1 mal redo: die zurückgenommenen Befehle $A_2...A_n$ werden erneut ausgeführt und in die Zustandsliste verschoben

Das Resultat ist wieder $A_1,B_1,B_2,A_2,...,A_n$.

Beispiel 2: Ersetzen eines Befehls

Es werden n Befehle $A_1...A_n$ ausgeführt. Anschließend soll A_2 durch B_1 und B_2 ersetzt werden. Dafür ist mit Yangs Triadic-Modell folgendes Vorgehen notwendig:

1. n-1 mal undo: wie Beispiel 1
2. command B_1, command B_2: ebenfalls wie Beispiel 1
3. 1 mal rotate: dreht die Rücknahmeliste, jetzt steht A_3 als erstes Element in der Liste
4. n-2 mal redo: alle anderen Befehle außer A_2 werden in ihrer ursprünglichen Reihenfolge wieder an die Zustandsliste angehängt

	Zustandsliste	Rücknahmeliste
vor 1.	$A_1,A_2,A_3,...,A_n$	
nach 1.	A_1	$A_2, A_3,...,A_n$
nach 2.	A_1,B_1,B_2	$A_2, A_3,...,A_n$
nach 3.	A_1,B_1,B_2	$A_3,A_4,...,A_n,A_2$
nach 4.	$A_1,B_1,B_2, A_3,...,A_n$	A_2

Abbildung 2.9: Inhalt der Ringlisten des Beispiels 2 (Triadic-Modell)

Im Vergleich zum Skript-Modell von Archer et al. hat das Triadic-Modell den Vorteil, mit drei Metabefehlen auszukommen. Gegenüber dem US&R-Modell von Vitter besteht der Vor-

teil, daß jeder Metabefehl eindeutig ist und somit keine weitere Interaktion mit dem Benutzer notwendig ist.
Allerdings führt die Möglichkeit, einzelne Befehle aus der Rücknahmeliste auszuführen oder an anderer Stelle einzufügen, dazu, daß aus der Liste eine ungeordnete Sammlung von Befehlen wird, aus der für den Benutzer nicht mehr ersichtlich ist, welche Befehle sequentiell hintereinander ausgeführt wurden (vgl. [Ber94]). Da die gesamte Bedienung des Modells doch eher ungewöhnlich für die meisten Benutzer ist, schlägt Yang eine Benutzungsschnittstelle vor, die das Triadic-Modell nur als Zwischenschicht benutzt und dem Anwender einfachere Bearbeitungsmöglichkeiten zur Verfügung stellt (vgl. [Yan88]). Hier stellt sich allerdings die Frage, ob es nicht sinnvoller ist, gleich ein anderes Undo-Modell zu verwenden. Wenn das Modell intern so kompliziert zu handhaben ist, daß man für eine vernünftige Verwendung eine weitere Benutzungsschicht benötigt, spricht vieles dafür, gleich auf ein Modell zurückzugreifen, daß der verwendeten Benutzungsschnittstelle entspricht.
Ein Problem, das Yang – wie übrigens auch Vitter – nicht anspricht, ist die Frage nach der Umkehrbarkeit von Befehlen. Er geht davon aus, daß jeder Befehl unabhängig von anderen Befehlen zurückgenommen werden kann. Dies kann leider nur in wenigen Fällen sichergestellt werden. Eine genauere Erläuterung dieser Problematik folgt in Kapitel 3.1.

2.6 Selektive Undo-Modelle

Die bisher vorgestellten nichtlinearen Undo-Modelle basierten alle auf der Idee, die Liste aller bisher ausgeführten Befehle als ein Skript zu betrachten. Durch die Ausführung eines Undo-Befehls soll ein Skript entstehen, das bis auf den zurückgenommenen Befehl identisch mit dem ursprünglichen Skript ist. Dies führt aber zu einer Verletzung der stabilen Ausführungseigenschaft (siehe Kapitel 2.2.4), da der nun nicht mehr vorhandene Befehl unter Umständen Einfluß auf seine nachfolgenden Befehle hatte. Diese Problematik wird von den Autoren der vorgestellten nichtlinearen Undo-Modelle entweder gar nicht behandelt, oder es wird zumindest keine Lösung dafür angeboten. Der Grund dafür ist, daß diese Modelle ihre Ursprünge in kommandozeilenorientierten Anwendungen haben, bei denen jeder Befehl isoliert vom aktuellen Zustand betrachtet werden konnte (vgl. [Ber94]). Beschäftigt man sich mit selektiven Undo-Modellen, wird schnell deutlich, daß der skriptbasierte Ansatz hierfür nicht geeignet ist. Selektive Undo-Modelle sind Modelle, die es erlauben, die Auswirkungen von einzelnen Befehlen aus der Befehlsgeschichte isoliert rückgängig zu machen.
Man stelle sich folgendes Beispiel vor: In einem Tabellenkalkulationsprogramm ändert der Benutzer den Wert einer Zelle von a nach b. Später möchte er diese Änderung rückgängig machen und den alten Wert wiederherstellen. In einem skriptbasierten Modell kann nicht sichergestellt werden, daß die Zelle am Ende tatsächlich den Wert a hat. Wenn in der Befehlsgeschichte weitere Änderungen an der entsprechenden Zelle vorhanden sind (Wert := c), werden diese den ursprünglichen Wert a wieder überschreiben, da sie später in dem Skript stehen. Wie man an diesem Beispiel sieht, können bei einem skriptbasierten Ansatz die Auswirkungen eines Undo-Befehls von später folgenden Befehlen, die auf das gleiche Objekt zugreifen, verdeckt werden.
Ein Undo-Modell, das diese Probleme nicht besitzt, ist das *direkte selektive Undo-Modell* (direct selective undo, vgl. [Ber94]). Hinter diesem Modell steckt folgende Idee: Statt von der gesamten Befehlsgeschichte sind die Auswirkungen eines Undo-Befehls nur von dem Befehl, der rückgängig gemacht werden soll, und dem aktuellen Zustand der Anwendung abhängig. Gegenüber den skriptbasierten Undo-Modellen, bei denen die Undo-Funktion immer ausgeführt werden kann, bietet das direkte selektive Undo-Modell die Möglichkeit, dem Anwender

zu signalisieren, daß die Rücknahme eines bestimmten Befehls keinen Sinn (mehr) macht (z.B. weil ein referenziertes Objekt nicht mehr existiert).
Ein Vorteil des selektiven Undo-Modells ist seine gegenüber anderen Modellen einfachere Benutzung sein: Einen einzelnen Befehl rückgängig zu machen soll möglichst auch nur eine einzelne Interaktion des Benutzers erfordern. Komplizierte Operation auf der Befehlsgeschichte, wie sie in den vorigen Kapiteln in verschiedenen Beispielen gezeigt wurden (vgl. Beispiel 1 und 2 jeweils in den Kapiteln 2.5.2 und 2.5.3), sind mit dem direkten selektiven Undo-Modell nicht mehr notwendig. Damit der Anwender von der erweiterten Undo-Funktion profitieren kann, muß es einfach und schnell möglich sein, den Befehl zu finden, der rückgängig gemacht werden soll.
Das direkte selektive Undo-Modell ist eine Erweiterung des linearen Undo-Modells mit Befehlsbaum (siehe Kapitel 2.4.2). Das heißt, es bietet zunächst einmal dessen kompletten Funktionsumfang. Zusätzlich besteht die Möglichkeit, beliebige Befehle – soweit sie selektives Undo unterstützen – rückgängig zu machen. Befehle, die selektives Undo unterstützten, müssen in der Lage sein, neue Befehle zu erzeugen, die genau die umgekehrten Effekte wie die ursprünglichen Befehle haben. Bei vielen Befehlen kann dies dadurch erreicht werden, daß die Parameter des entsprechenden Befehls ausgetauscht werden. Diese umgekehrten Befehle werden an das Ende der Befehlsgeschichte gehängt und heben so die Wirkung des ursprünglichen Befehls auf.

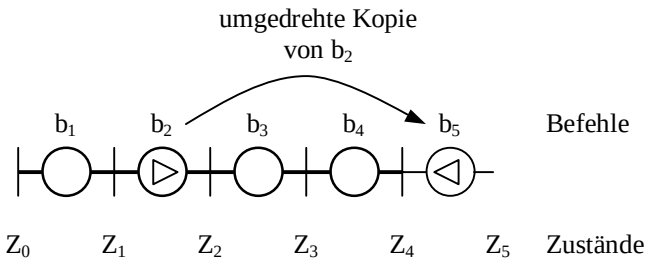

Abbildung 2.10: Rückgängigmachen eines Befehls mit einer „umgedrehten“ Kopie

Beispielsweise besteht der umgekehrte Befehl zu einer Kontoabbuchung aus einer Rückbuchung, die das Saldo des Kontos wieder um den gleichen Betrag erhöht. Falls es durch die Buchung Seiteneffekte, wie z.B. erhobene Gebühren, gibt, müssen diese ebenfalls durch eine Gegenbuchung egalisiert werden. Die Auswirkungen einer zu einem früheren Zeitpunkt ausgeführten Buchung (b_2 in Abbildung 2.10) werden rückgängig gemacht, indem der zu dieser Buchung umgekehrte Befehl (die Rückbuchung um den gleichen Betrag, b_5) ausgeführt wird. Dieser wird wie jeder normale Befehl in der Befehlsgeschichte abgespeichert.
Neben der selektiven Undo-Funktion besitzt dieses Modell auch eine selektive Redo-Funktion (vgl. [Ber93]). Wie bereits erwähnt, nutzt das selektive Undo-Modell das lineare Undo-Modell mit Befehlsbaum für die normale lineare Undo-Funktion. Bei der Benutzung dieser linearen Undo-Funktion entsteht ein Speicherbaum, der neben dem aktuellen Zweig der Befehlsgeschichte aus weiteren Zweigen bestehen kann, die alle im Laufe der Bearbeitung durch die lineare Undo-Funktion wieder zurückgenommen wurden. Aus diesen Zweigen des Befehlsbaums kann der Benutzer einen beliebigen Befehl auswählen und ihn erneut ausführen. Dies führt dazu, daß eine Kopie des entsprechenden Befehls an das Ende der Befehlsgeschichte gehängt wird (siehe Abbildung 2.11).
Für den Entwickler hat das direkte selektive Undo-Modell den Vorteil, daß die Undo-Funktion für jede Befehlsklasse unabhängig von anderen programmiert werden kann. Somit muß die Funktion nicht für alle Befehle, sondern nur für diejenigen, bei denen der Bedarf besteht, entwickelt werden.

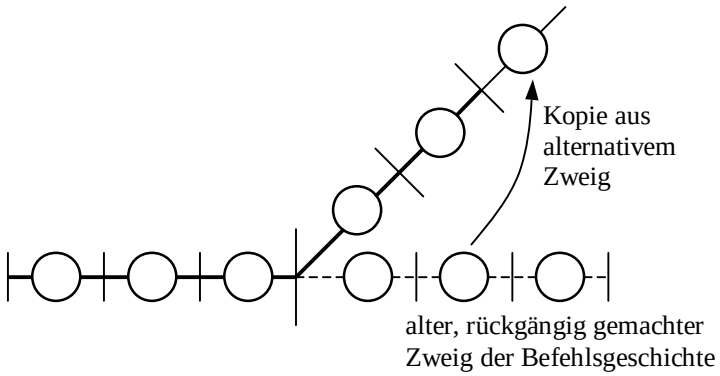

Abbildung 2.11: Selektives Redo durch Kopie aus einem alternativem Zweig

Ein weiterer Vorteil des direkten selektiven Undo-Modells liegt in seiner flexiblen Einsetzbarkeit. Durch die Benutzung eines baumbasierten linearen Undo-Modells werden die Standardfälle von einfachen linearen Undo- und Redo-Funktionen abgedeckt. Die Möglichkeit, einen Befehl selektiv, also außerhalb seiner ursprünglichen Reihenfolge, rückgängig zu machen oder erneut auszuführen, können Befehle zusätzlich anbieten. Werden selektive Undo- und Redo-Funktionen nicht angeboten – entweder weil sie nicht implementiert sind oder weil ein Implementierung gar nicht möglich ist –, so muß lediglich sichergestellt werden, daß der Benutzer diese Funktionen für die jeweiligen Befehle von der Anwendung nicht angeboten bekommt. Anderenfalls müßte man den Benutzer jedesmal mit einer Fehlermeldung konfrontieren oder ihn gar ganz darüber im unklaren lassen, daß die gewünschte Aktion nicht ausgeführt wurde. Beides widerspricht dem Benutzungsmodell moderner interaktiver Programme: Der Benutzer sollte davon ausgehen können, daß er eine ihm zur Verfügung stehende Funktion auch verwenden darf. Wenn allerdings nicht einmal ersichtlich ist, ob die Funktion überhaupt ausgeführt wurde, dann ist sie für den praktischen Einsatz nicht verwendbar.
Problematisch bei dieser Lösung ist, daß zwei unterschiedliche Benutzungsmodelle aufeinandertreffen und somit das Gesamtbenutzungsmodell unklar wird. Alle Aktionen, die mit Hilfe der selektiven Undo- oder Redo-Funktion ausgeführt wurden, führen dazu, daß an das Ende der Befehlsgeschichte ein neuer Befehl angehängt wird. Von der linearen Undo/Redo-Funktion wird dieser Befehl genauso wie jeder andere behandelt. Das bedeutet, daß aus einem selektiven Undo, das mit Hilfe einer umgedrehten Kopie ausgeführt wurde, ein Redo wird, wenn auf diesem das lineare Undo ausgeführt wird.
Dazu folgendes Beispiel: Ein Benutzer führt in einer Anwendung mit direktem selektiven Undo die Befehle a, b und c aus und macht anschließend den Befehl b mit der selektiven Undo-Funktion rückgängig. Für den Benutzer sichtbar ist jetzt nur noch das Ergebnis der Befehle a und c. Tatsächlich wurde aber a b c ~b ausgeführt (~b ist die Umkehrfunktion von b). Was ist nun das Ergebnis, wenn die normale (lineare) Undo-Funktion aufgerufen wird? Erwarten würde der Benutzer nur noch das Resultat von Befehl a. Das Ergebnis ist aber, daß lediglich das selektive Undo rückgängig gemacht wurde und somit die Befehle a, b und c ausgeführt wurden. Ähnliche Fällen lassen sich für viele Kombinationen von selektiven und linearen Undo- und Redo Funktionen konstruieren.
Um Fehlbedienungen zu vermeiden, müssen daher neben den normalen Aktionen auch alle selektiven Undo-Schritte in der Befehlsgeschichte aufgeführt werden. Erst dadurch wird für den Benutzer ersichtlich, auf welchen Befehl sich die lineare Undo-Funktion jeweils bezieht. Die Undo- oder Redo-Schritte der linearen Undo-Funktion können dagegen modellbedingt gar nicht in der Befehlsgeschichte aufgeführt werden. Somit besteht hier eine Diskrepanz zwischen den beiden einzelnen Benutzungsmodellen, die das Gesamtbenutzungsmodell schwer verständlich erscheinen läßt.

2.7 Zusammenfassung

Neben den Grundlagen von Undo-Funktionen wurden in diesem Kapitel verschiedene Undo-Modelle vorgestellt und ihre Vor- und Nachteile analysiert. Das einfachste der vorgestellten Modelle ist das Single-Undo-Modell. Es erlaubt nur, zwischen den letzten beiden Zuständen einer Anwendung hin- und herzuwechseln. Dieses Modell läßt sich zwar relativ einfach implementieren, ist aber aufgrund seiner Beschränkung auf einen Undo-Schritt für die meisten Anwendungen schlecht geeignet. In Programmen mit graphischer Oberfläche werden falsche Bedienungsschritte oft nicht sofort, sondern erst nach weiteren Arbeitsschritten erkannt. Mit dem Single-Undo-Modell ist eine Korrektur dann nicht mehr möglich.
Lineare Undo-Modelle zeichnen sich dadurch aus, daß die Anzahl der Aktionen, die rückgängig gemacht werden können, nur durch die Größe des für die Befehlsgeschichte reservierten Speicherplatzes begrenzt ist. Bei dem bekanntesten linearen Undo-Modell, dem eingeschränkten linearen Undo-Modell, sind allerdings nur die Zustände wiederherstellbar, die direkte Vorgänger des aktuellen Zustands sind. Sämtliche Aktionen, die zurückgenommen wurden und durch andere Aktionen ersetzt worden sind, können nicht wiederhergestellt werden. Das weniger verbreitete baumbasierte lineare Undo-Modell erlaubt es, auch diese wiederherzustellen. Aber auch hier kann nur jeweils der komplette Zweig der Befehlsgeschichte rekonstruiert werden. Später durchgeführte Änderungen eines anderen Zweigs der Befehlsgeschichte sind somit verloren. Da beide Undo-Modelle die stabile Ausführungseigenschaft erfüllen (kein Befehl wird in einem anderen als seinem Ursprungszustand ausgeführt), muß sich der Entwickler nicht um Probleme mit Befehlen kümmern, die aufgrund veränderter Bedingungen nicht mehr korrekt oder gar nicht mehr ausgeführt werden können.
Aktuelle Programme, die eine Undo-Unterstützung anbieten, nutzen dabei fast ausschließlich Lösungen, die auf dem beschränkten linearen Undo-Modell basieren. Daher dürften die meisten Anwender sofort mit einer Undo-Funktion, die diesem Modell entspricht, vertraut sein. Das baumbasierte lineare Undo-Modell ist dagegen in der Praxis kaum anzutreffen und benötigt zudem eine wesentlich komplexere Benutzungsschnittstelle. Daher ist das beschränkte lineare Undo-Modell vorzuziehen.
Die nichtlinearen Undo-Modelle leiden alle an einer gemeinsamen Schwäche: Sie gehen davon aus, daß Befehle in einem anderen als ihrem ursprünglichen Zustand ausgeführt werden können. Die Ursache dafür liegt darin, daß diese Modell entstanden, als noch kommandozeilenorientierte Anwendungen vorherrschend waren.
Die Nichterfüllung der stabilen Ausführungseigenschaft führt für den Entwickler zu erheblichem Mehraufwand. Es muß die Vorbedingung geprüft werden, ob ein Befehl noch ausführbar ist, und es müssen für den negativen Fall passende Fehlermeldungen bereitgehalten werden. Für den Benutzer liegt die Problematik darin, daß die Ergebnisse der Undo- und Redo-Funktionen immer weniger vorhersehbar sind, je öfter diese Funktionen benutzt wurden. Da oft Abhängigkeiten zwischen vorhergehenden und nachfolgenden Befehlen bestehen, ist nicht mehr vorhersehbar, ob das Ergebnis einer Undo- oder Redo-Funktion wirklich das gewünschte ist. Zudem besitzen alle drei vorgestellten Modelle ein gewöhnungsbedürftiges Benutzungsmodell, das einen gewissen Einarbeitungsaufwand erfordert, um den vollen Funktionsumfang nutzen zu können. Insgesamt erscheinen die drei vorgestellten nichtlinearen Undo-Modelle wenig geeignet für die Implementierung eines Prototypen einer allgemeinen Undo-Funktion.
Das uneingeschränkte lineare Undo-Modell besitzt ebenfalls die genannten Nachteile. Zusätzlich ist es aber in seiner Funktionalität eingeschränkt. Beliebige Änderungen an der Befehlsgeschichte, die mit den nichtlinearen Modellen – wenn auch oft nur mit vielen Schritten – möglich sind, können mit diesem Modell nicht durchgeführt werden. Dieses Modell vereint daher Unflexibilität und schwierige Implementierbarkeit miteinander.

Als letztes wurde in diesem Kapitel das direkte selektive Undo-Modell vorgestellt. Da dieses Modell auf dem linearen Undo-Modell mit Befehlsbaum aufbaut, besitzt es zunächst auch dessen Vor- und Nachteile. Zusätzlich erlaubt dieses Modell mit Hilfe seiner selektiven Undo-Funktion für alle Befehle, die diese implementieren, beliebige Änderungen an der Befehlsgeschichte vorzunehmen. Von den Undo-Modellen, die die stabile Ausführungseigenschaft nicht erfüllen, ist dies das am flexibelsten verwendbare.
Zusammenfassend betrachtet erscheint das eingeschränkte lineare Undo-Modell am besten geeignet für die Entwicklung einer allgemeinen Undo-Funktion, da es einen guten Kompromiß aus Funktionsumfang und Entwicklungsaufwand darstellt. Zudem sind die meisten Anwender mit diesem Modell schon vertraut, da in modernen Programmen nahezu ausschließlich dieses Modell verwendet wird.
Für eine spätere Erweiterung des Funktionsumfangs kommt das lineare Undo-Modell mit Befehlsbaum oder das direkte selektive Undo-Modell in Frage. Letzteres ist das optimale Undo-Modell, wenn man die Probleme bei der Entwicklung nicht betrachtet. In der folgenden Tabelle werden die Vor- und Nachteile der einzelnen Modelle noch einmal zusammengefaßt:

	mehr-schrittiges Undo	Undo nach neuen Befehlen	stabile Ausfüh-rungseigen.	Bedien-/ Erlern-barkeit	Funktions-umfang	Verbrei-tung	Entwick-lungs-aufwand
Single-Undo-Modell	nein	nein	ja	++	--	o	++
History-Undo-Modell	ja	ja	ja	-	o	-	+
eingeschr. lineares UM	ja	nein	ja	++	-	++	+
lineares UM + Bef'baum	ja	ja	ja	+	o	--	o
normales lineares UM	ja	nein	nein	+	o	--	-
allgem. Skript-Modell	ja	ja	nein	--	++	--	--
US&R-Modell	ja	ja	nein	o	+	--	-
Triadic-Undo-Modell	ja	ja	nein	-	+	--	-
direktes selektives UM	ja	ja	nein	+	++	--	-

Abbildung 2.12: Vergleich und Bewertung der vorgestellten Undo-Modelle

3 Beispiele für die Realisierung von Undo

Nachdem im vorigen Kapitel verschiedene Modelle, auf denen Undo-Funktionen basieren können, vorgestellt wurden, beschäftigt sich dieses Kapitel mit der praktischen Realisierung dieser Funktionen. Zunächst werden dazu vier allgemeine Strategien vorgestellt, mit denen eine Anwendung in einen vorhergehenden Zustand zurückversetzt werden kann. Anschließend werden die Architektur und der Programmcode von drei Beispielen analysiert und miteinander verglichen. Die drei Beispiele unterscheiden sich im Grad ihrer Abstraktion: Das erste Beispiel, eine Architektur, die nur das Befehlsmuster als Grundlage hat, erfordert vom Entwickler, einen Großteil des Programmcodes für die Undo-Funktion selbst zu implementieren, ist dafür aber universell einsetzbar. Beim zweiten Beispiel handelt es sich um das Undo-Paket aus der Java-Swing-Bibliothek. Wie auch beim Einsatz anderer Pakete aus Swing erspart man es sich hierbei, einen Großteil des Programmcodes selbst zu implementieren. Dafür ist man aber auf Programme, die diese Bibliothek benutzen, beschränkt. Das dritte Beispiel zeigt die Undo-Funktion des JHotDraw-Rahmenwerks. Diese Undo-Funktion erfordert die geringsten Anpassungen am eigenen Code, falls Anwendungen, die auf dem JHotDraw-Rahmenwerk basieren, eingesetzt werden. Damit beschränkt sich allerdings der Einsatzbereich auf JHotDraw-basierte Zeichenprogramme. Alle drei hier vorgestellten Lösungen basieren auf dem eingeschränkten linearen Undo-Modell.

3.1 Wiederherstellungsstrategien

Wiederherstellungsstrategien beschreiben, auf welche technische Art und Weise der Zustand einer Anwendung rekonstruiert werden kann, wenn Undo oder Redo ausgeführt werden soll. Es kann dabei zwischen vier verschiedenen Strategien unterschieden werden (vgl. [Arc84]).
Die einfachste Strategie ist die *Strategie des kompletten erneuten Ausführens* (complete rerun strategy). Bei dieser Strategie wird zunächst mitprotokolliert, welche Schritte ausgeführt wurden. Wenn die Undo-Funktion ausgeführt werden soll, wird die Anwendung in einen bekannten Zustand (z.B. den Startzustand der Anwendung) zurückversetzt, und es werden alle protokollierten Schritte, mit Ausnahme des letzten, erneut nacheinander abgearbeitet. Dieses Vorgehen ist dadurch charakterisiert, daß es um so ineffizienter wird, je mehr Schritte zuvor ausgeführt wurden. Neben dem Rechenaufwand für die erneute Ausführung aller zuvor getätigter Arbeitsschritte ist mit ihrer steigenden Anzahl auch der benötigte Speicherplatz ein Problem, da es nicht möglich ist, Teile des Protokolls zu löschen. Aus den genannten Gründen ist diese Strategie für interaktive Anwendungen nicht geeignet. Für Anwendungen, die relativ selten die Wiederherstellungsfunktion benutzen, eignet sich diese Strategie, wenn zwischenzeitlich der komplette Zustand der Anwendung gesichert werden kann und somit zumindest hin und wieder ein neuer Ausgangszustand vorliegt.
Eine zweite mögliche Strategie ist die *Strategie der vollständigen Sicherung* (full checkpoint strategy). Bei dieser Strategie wird nach jeder ausgeführten Aktion der komplette Zustand der Anwendung gesichert. Ein Undo erfolgt, indem die Anwendung in den zuvor gesicherten Zustand zurückversetzt wird. Die Sicherung und die Wiederherstellung lassen sich relativ einfach generisch lösen, so daß für alle Aktionen die gleiche Undo-Methode benutzt werden kann. Allerdings erfordert auch diese Strategie bei größeren Anwendungen sehr viel Rechenaufwand für die Sicherung des unter Umständen umfangreichen Zustands der Anwendung. Der Speicherplatzverbrauch dieser Strategie ist auch bei kleinen Anwendungen schon so groß, daß die Benutzung nur für Anwendungen, die sehr selten ihren Zustand sichern, sinnvoll erscheint.

Die am häufigsten eingesetzte Strategie ist die *Strategie der partiellen Sicherung* (partial checkpoint strategy). Da bei der Ausführung eines Arbeitsschritts meist nur kleine Teile einer Anwendung (einzelne Variablen oder Objekte) verändert werden, reicht es im allgemeinen aus, nur diese zu rekonstruieren. Dies läßt sich zwar relativ schnell und ohne großen Speicherplatzverbrauch realisieren, erfordert aber Detailwissen über die Anwendung. Oft werden durch Seiteneffekte Änderungen (absichtlich oder unabsichtlich) hervorgerufen, die auf den ersten Blick nicht ersichtlich sind. Werden die durch die Seiteneffekte veränderten Werte nach einem Undo nicht mitrekonstruiert, kann die Anwendung in einen inkonsistenten Zustand geraten, der einen weiteren Programmablauf unmöglich macht. Außerdem muß bei dieser Strategie für nahezu jede Aktion eine eigene Methode zur Rekonstruktion entwickelt werden, da selten exakt die gleichen Variablen oder Objekte gesichert und später rekonstruiert werden sollen.
Die letzte Strategie ist die *Strategie des inversen Befehls* (inverse command strategy). Viele mathematische Operationen zeichnen sich dadurch aus, daß es zu ihnen eine Umkehrfunktion (eine Funktion, die genau die gegenteilige Wirkung ihrer zugehörigen Funktion hat) gibt. Wenn eine Aktion nur aus solchen Operationen besteht, reicht es aus, die passenden Umkehrfunktionen auszuführen, um den ursprünglichen Zustand wiederherzustellen. Eine Sicherung von Zustandswerten ist dann nicht nötig. Leider sind im allgemeinen Befehle wie z.B. x := x + 1, für die eine Umkehrfunktion gebildet werden kann, gegenüber Befehle wie etwa x := y, die eine Sicherung von Werten erfordern, eher eine verschwindend geringe Minderheit. Daher kann diese Strategie auch nur selten zur Anwendung kommen.
Die drei in diesem Kapitel vorgestellten Beispielarchitekturen sind für die Strategie der partiellen Sicherung ausgelegt. Der Entwickler muß bei allen Architekturen für jede Aktion die Variablen und Objekte bestimmen, die rekonstruiert werden sollen, wenn Undo oder Redo ausgeführt werden. Diese werden dann, je nach Architektur, auf unterschiedliche Weise gespeichert. Prinzipiell könnte bei allen Architekturen auch die Strategie des inversen Befehls verwendet werden, wenn für eine Aktion eine passende Umkehrfunktion existiert.

3.2 Undo mit Hilfe von Befehlsobjekten

Entwurfsmuster sind in der Softwareentwicklung allgemein verwendbare Lösungen für häufig wiederkehrende gleiche oder ähnliche Entwurfsprobleme. Sie geben eine allgemeine Struktur zur Lösung des Problems vor, identifizieren dabei die beteiligten Klassen und Objekte und beschreiben ihre Rollen und Aufgaben samt der dazu nötigen Interaktion. Zudem beschreiben Entwurfsmuster unter welchen Voraussetzungen sie einsetzbar sind, welche Konsequenzen ihr Einsatz hat und wie sie am besten zu implementieren sind (vgl. [Gam96]).
Mit Hilfe eines solchen Entwurfsmusters, dem *Befehlsmuster* (command pattern), läßt sich auf sehr einfache Weise eine allgemein verwendbare Undo-Funktion realisieren. In den folgenden beiden Abschnitten wird der allgemeine Aufbau des Befehlsmusters und wie sich damit eine Undo-Funktion entwickeln läßt, beschrieben. Eine ausführliche Beschreibung des Befehlsmusters liefern Gamma et al. in ihrem Buch über Entwurfsmuster (siehe [Gam96]).

3.2.1 Das Befehlsmuster

Die Kernidee des Befehlsmusters besteht darin, jeden Befehl einer Anwendung in einer eigenen Klasse zu kapseln. Befehle sind hierbei einzelne oder mehrere zusammenhängende Operationen, die den Zustand einer Anwendung ändern. Jede dieser Befehlsklassen implementiert dabei die Schnittstelle (oder erbt alternativ von der abstrakten Klasse) eines allgemeinen Befehls. In dieser Schnittstelle ist eine Methode `führeAus()` deklariert, die für den Start der

eigentlichen Programmlogik des Befehls sorgt. Die Programmlogik kann entweder in der Befehlsklasse selbst oder in einer weiteren Klasse, die der Befehlsklasse bekannt ist, implementiert sein.

Die Teilnehmer am Befehlsmuster sind:

- eine allgemeine Befehlsschnittstelle, die eine Ausführungsmethode definiert
- ein konkreter Befehl, der diese Methode durch Aufrufen einer Operation am Empfänger implementiert
- ein Klient, der das konkrete Befehlsobjekt erzeugt und ihm dabei eine Referenz auf den Empfänger übergibt; dem Aufrufer übergibt er eine Referenz auf dieses Befehlsobjekt
- ein Empfänger, der die vom Befehlsobjekt an ihm aufgerufene Methode ausführt
- ein Aufrufer, der die Anfrage am Befehlsobjekt auslöst; er kennt nur die allgemeine Befehlsschnittstelle

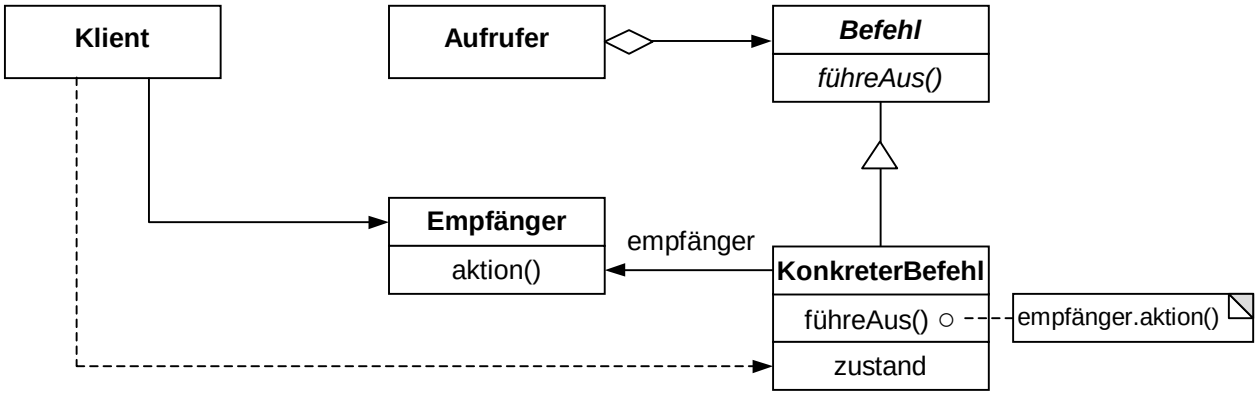

Abbildung 3.1: Struktur des Befehlsmusters

Die denkbaren Anwendungsfälle für das Befehlsmuster sind vielfältig:

- Mit Hilfe des Befehlsmusters können Anfragen an Objekte, von denen die genaue Schnittstelle unbekannt ist, gestellt werden. Dabei kann unbekannt sein, wie der Name der auszuführenden Methode ist, oder über das Objekt selbst liegen keine Informationen vor. Dieser Fall tritt zum Beispiel bei Klassenbibliotheken für Benutzungsschnittstellen auf. Die Steuerungselemente müssen als Reaktion auf eine Eingabe des Benutzers Anfragen an Anwendungsobjekte stellen, die den Entwicklern der Bibliothek unbekannt sind. Hier wird das Befehlsmuster verwendet, um das die Anfrage auslösende Objekt von dem Objekt zu trennen, das weiß, wie sie auszuführen ist.
- Das Befehlsmuster wird zudem verwendet, um aus primitiven komplexere Operationen aufzubauen. Mit Hilfe des Befehlsmusters werden Transaktionen modelliert, die eine Menge von primitiven Operationen kapseln. Da alle Befehlsobjekte die gleiche Schnittstelle besitzen, ist es möglich, alle Transaktionen auf die gleiche Weise aufzurufen.
- Ein weiterer Anwendungsfall des Befehlsmusters liegt vor, wenn Anfragen zu einem späteren Zeitpunkt ausgeführt werden sollen, als sie spezifiziert werden. Das Befehlsobjekt kann unabhängig von der ursprünglichen Anfrage bestehen bleiben und kann somit verzögert und unter Umständen sogar auf einem anderen als dem aufrufenden Rechner ausgeführt werden.

- Mit Hilfe des Befehlsmusters kann ein Protokollmechanismus realisiert werden, der Änderungen im System speichert und im Falle eines Absturzes den letzten Zustand der Anwendung durch erneutes Ausführen dieser Änderungen wiederherstellt. Realisiert wird dies, indem die Befehlsobjekte auf einem persistenten Speichermedium, wie z.B. einer Festplatte, gespeichert werden, um im Bedarfsfall wieder eingelesen werden zu können.
- Schließlich ist es mit Hilfe des Befehlsmuster möglich, eine Undo- und Redo-Funktion zu entwickeln.

3.2.2 Realisierung von Undo-Funktionen mit Hilfe des Befehlsmusters

Das Konzept des Befehlsmusters, jeden Befehl in ein eigenes Objekt zu kapseln, eignet sich hervorragend, um auf einfache Weise eine Undo/Redo-Funktion zu realisieren (vgl. [Mey88]). Dazu wird die allgemeine Befehlsschnittstelle um eine `undo()`-Methode erweitert, die, wie die `führeAus()`-Methode, von jeder konkreten Befehlsklasse implementiert werden muß. Diese `undo()`-Methode sorgt für die Rekonstruktion des ursprünglichen Zustands im Empfängerobjekt, indem sie den veränderten Werten und Objekten die zuvor gesicherten Daten zuweist oder eine Operation ausführt, die genau die umgekehrten Auswirkungen, wie die `führeAus()`-Methode hat.
Die Redo-Funktion, die das Ergebnis der Undo-Funktion rückgängig machen soll, kann entweder durch den erneuten Aufruf der `führeAus()`-Methode oder durch eine eigene `redo()`-Methode ausgelöst werden. Eine explizite `redo()`-Methode hat gegenüber der Mitverwendung der `führeAus()`-Methode den Vorteil, daß bei jeder erneuten Ausführung der Zustand des Empfängers nicht noch einmal gesichert werden muß.
Die `führeAus()`-Methode jeder Befehlsklasse muß so erweitert werden, daß sie den Zustand des Empfängers vor der Ausführung der eigentlichen Operation sichern kann. Jedes Befehlsobjekt muß eine Referenz auf das Empfängerobjekt, die Argumente für die vom Empfänger auszuführenden Operationen und alle Werte und Objekte, die sich im Empfänger bei der Ausführung der Operation ändern können, speichern. Das Empfängerobjekt muß dem Befehl Methoden zur Verfügung stellen, die es ihm ermöglichen, auf die zu sichernden Werte und Objekte zuzugreifen, um sie abspeichern und wiederherstellen zu können.
Mit diesen Zugriffsmethoden wird allerdings auch die fachliche Kapselung des Empfängerobjekts durchbrochen. Neben den Befehlsobjekten können mit Hilfe dieser Methoden auch andere Objekte an den dafür vorgesehenen fachlichen Methoden vorbei direkt auf die interne Repräsentation der Daten des Empfängerobjekts zugreifen. Damit wird eines der Grundprinzipien der objektorientierten Programmierung verletzt. Wenn die Rolle des Empfängers und die Rolle des Befehls in einem einzigen Objekt zusammengefaßt werden, besteht diese Problematik nicht, da dann die Zugriffsmethoden nicht veröffentlicht werden müssen.
Die meisten Befehlsobjekte werden während eines Programmablaufs mehrfach benutzt. Um Änderungen an den zuvor gesicherten Daten zu vermeiden, muß jedes Befehlsobjekt nach dem Aufruf der `führeAus()`-Methode mit Hilfe einer tiefen Kopie dupliziert werden. (Eine tiefe Kopie eines Objekts ist eine Kopie, bei der neben den primitiven Werten auch alle Objekte, die direkt oder indirekt über Referenzen erreichbar sind, dupliziert werden.) Alternativ kann, um Seiteneffekte zu vermeiden, für jeden Befehl vor der Ausführung ein neues Objekt erzeugt werden, das anschließend gespeichert wird.
Die Sicherung der ausgeführten Befehlsobjekte erfolgt in einer Liste, der Befehlsgeschichte. Die maximale Länge dieser Liste legt fest, wie viele Undo- bzw. Redo-Schritte möglich sind.

3.2.3 Bewertung

Auf dem Befehlsmuster basierende Undo-Funktionen zeichnen sich durch ihren einfachen Aufbau und ihre unkomplizierte Realisierbarkeit aus. Der für das Rückgängigmachen notwendige Programmcode befindet sich nur im Befehlsobjekt und somit nur an einer einzigen Stelle. Da sich im Befehlsobjekt die Programmlogik befindet, die für die Ausführung der eigentlichen Operation zuständig ist, ist dies auch die am besten geeignete Stelle für die Undo-Programmlogik. Der Entwickler eines Befehlsobjekts weiß selbst am besten, wie die Wirkung „seines“ Befehls rückgängig gemacht werden kann.
Wenn es sich beim Empfänger und beim Befehl um verschiedene Objekte handelt, besteht der Nachteil, daß Zugriffsmethoden auf Interna des Empfängers öffentlich gemacht werden müssen. Ein weiterer Nachteil dieses Undo-Konzepts liegt darin, daß nur wenige Programme von Anfang an mit Hilfe des Befehlsmusters entwickelt werden. Um solche Programme nachträglich um eine Undo-Funktion zu erweitern, ist daher eine komplette Änderung des Programmaufbaus notwendig. Bei Anwendungen, die von Grund auf neu entwickelt werden, besteht dieses Problem dagegen nicht. Daher sollte möglichst schon beim Entwurf einer Anwendung die Entscheidung fallen, ob sie, wenn vielleicht auch erst später, Undo unterstützen soll.
Nahezu alle objektorientierten Programme mit einer Undo-Funktion, von denen der Quellcode zugänglich ist, benutzen das Befehlsmuster oder ein auf diesem aufbauendes Konzept für die Realisierung ihrer Undo-Funktion. Dies gilt auch für die im folgenden vorgestellten Beispiele.

3.3 Undo-Unterstützung in Swing

Swing ist eine Sammlung von Komponenten zur Erstellung von Anwendungen mit graphischer Benutzungsoberfläche für Java (siehe [Eck98]). Es erlaubt die Entwicklung von Programmen, die plattformübergreifend lauffähig sind und je nach eingesetztem Betriebssystem dessen typisches Aussehen und dessen typische Handhabung (Look & Feel) besitzen. Swing basiert auf der Java-Standardklassenbibliothek für graphischen Benutzungsoberfläche, dem „Abstract Window Toolkit“ (AWT), ist im Gegensatz zu diesem aber vollständig in Java geschrieben. Es kann somit auf jedem System eingesetzt werden, für das eine virtuelle Java-Maschine existiert. Swing ist, wie das AWT, Bestandteil der „Java Foundation Classes“. Es handelt sich hierbei um eine Sammlung von Bibliotheken, die Entwickler bei Erstellung von Java-Programmen mit anpaßbaren Komponenten unterstützen sollen. Swing wurde ursprünglich als eigenständige Ergänzung zum Java Development Kit (JDK) 1.1 entwickelt. (Das JDK ist die von Sun Microsystems entwickelte Referenzentwicklungsumgebung für Java.) Seit der Version 1.2 des JDK ist es dessen fester Bestandteil und somit auf allen Systemen mit aktueller Java-Unterstützung verfügbar.

3.3.1 Das Swing-Undo-Paket

Neben Komponenten zur Gestaltung von Oberflächen beinhaltet Swing auch Pakete, die funktionale Erweiterungen von Programmen erleichtern. Eines dieser Pakete ist das `javax.swing.undo`-Paket (siehe [Sun02]). Es handelt sich hierbei um eine Sammlung von Klassen, die es ermöglichen, Java-Programme mit Undo/Redo-Unterstützung auszustatten. Die Verwendung der Swing-Klassen zur Erstellung der graphischen Benutzungsoberfläche ist nicht zwingend notwendig, um das `javax.swing.undo`-Paket einsetzen zu können.
Mit Hilfe dieses Pakets erstellte Undo-Funktionen basieren normalerweise auf dem eingeschränkten linearen Undo-Modell (siehe Kapitel 2.4.1). Als Sonderfall ist es aber auch mög-

lich, Anwendungen mit einer Undo-Funktion zu versehen, die dem Single-Undo-Modell (siehe Kapitel 2.3.1) entspricht.
Das `javax.swing.undo`-Paket benutzt eine abgewandelte Form des Befehlsmusters, um die Undo- und Redo-Funktionalität zu realisieren. Wie in Kapitel 3.2.1 beschrieben, wird jeder Befehl in einer eigenen Klasse gekapselt. Allerdings wird nicht, wie bei Gamma beschrieben (siehe [Gam96]), für jeden einzelnen Befehl ein neues Befehlsobjekt angelegt und dieses gespeichert, sondern nur ein Exemplar von jeder Befehlsklasse. In dieser Befehlsklasse sind die Methoden implementiert, die für die Ausführung des Befehls notwendig sind. Zusätzlich wird in dieser Befehlsklasse bei jeder Ausführung ein Änderungsobjekt (`UndoableEdit`) erzeugt, in dem die Auswirkungen dieses Befehls und die Implementierung der Undo- und der Redo-Methode gespeichert sind. Dieses Objekt wird in der Befehlsgeschichte, die durch den `UndoManager` verwaltet wird, gespeichert. Es dient dazu, später den Zustand vor (bei Undo)

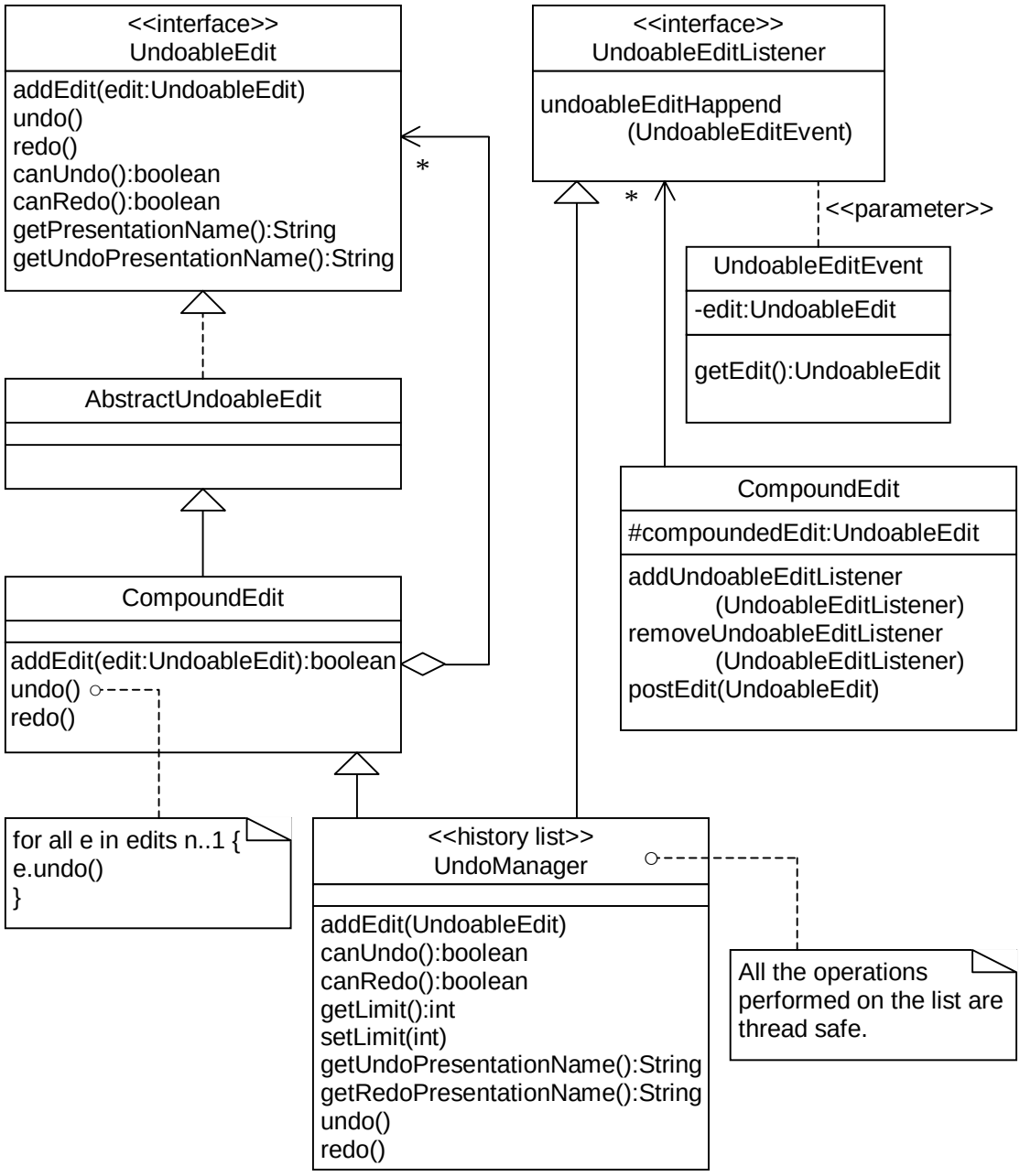

Abbildung 3.2: Klassendiagramm des Swing-Undo-Konzepts (aus [Mes98])

oder nach (bei Redo) der Ausführung des Befehls rekonstruieren zu können. An die Stelle der Befehlsklasse tritt eine Unterklasse des `EventListeners`, der `UndoableEditListener`.
Das in Abbildung 3.2 dargestellte Klassendiagramm stellt die Struktur der am Swing-Undo-Konzept beteiligten Klassen und Schnittstellen dar. Sämtliche Klassen und Schnittstellen sind Bestandteil des Paketes `javax.swing.undo`. Ausgenommen hiervon sind das Interface `UndoableEditListener` und die Klasse `UndoableEditEvent`, die zum Paket `javax.swing.event` gehören.

3.3.2 Klassen und Schnittstellen des Swing-Undo-Konzepts

Im folgenden Abschnitt werden die Klassen und Schnittstellen des `javax.swing.undo`-Pakets und ihre wichtigsten Methoden vorgestellt.

Die Schnittstelle UndoableEdit

Wie bereits zuvor beschrieben, wird für jeden ausgeführten Befehl ein Objekt erzeugt, in dem die Änderungen, die dieser Befehl verursacht hat, gespeichert sind. `UndoableEdit` ist die Schnittstelle, die jedes dieser Änderungsobjekte implementieren muß, damit es später möglich ist, diesen Befehl rückgängig zu machen.
Die wichtigsten Methoden dieser Schnittstelle sind `undo()` und `redo()`. In diesen Methoden muß implementiert werden, wie die Änderungen, die in diesem Objekt gespeichert sind, rückgängig gemacht bzw. wiederhergestellt werden können. Die dazu nötigen Daten (einzelne Variablen oder Objekte) werden bei der Erzeugung des `UndoableEdit`-Objekts als Parameter des Konstruktors übergeben und in diesem gespeichert.
Daneben gibt es die sondierenden Methoden `canUndo()` und `canRedo()`. Sie erlauben die Abfrage, ob es bei diesem `UndoableEdit`-Objekt im aktuellen Zustand möglich ist `undo()` bzw. `redo()` auszuführen. Wenn dies nicht möglich ist, können dafür verschiedene Gründe vorliegen:

- Die `undo()`- oder die `redo()`-Methode des UndoableEdit-Objekts sind nicht implementiert.
- Die `die()`-Methode wurde an diesem Objekt aufgerufen, um kenntlich zu machen, daß ein Undo oder Redo aus externen Gründen nicht mehr möglich ist.
- Der Befehl wurde bereits zurückgenommen (kein Undo möglich).
- Der Befehl wurde noch nicht zurückgenommen (kein Redo möglich).

Mit `isSignificant()` kann abgefragt werden, ob diese Änderung für den Benutzer als eigener Arbeitsschritt angezeigt wird oder ob sie nur als Seiteneffekt eines anderen Befehls ausgeführt wird (beispielsweise die Rekonstruktion einer Selektion).
Schließlich besteht die Möglichkeit abzufragen, unter welchem Namen eine Änderung und deren Undo- und Redo-Funktion an der Oberfläche angezeigt wird.
Beispiel: `getPresentationName()`: „Löschen“
`getUndoPresentationName()`: „Rückgängig: Löschen“
`getRedoPresentationName()`: „Wiederherstellen: Löschen“

Die Klasse AbstractUndoableEdit

Diese Klasse stellt eine Standardimplementierung der `UndoableEdit`-Schnittstelle bereit. Trotz des Namens handelt es sich nicht um eine abstrakte Klasse. Für jede Methode der Schnittstelle ist eine Standardimplementierung vorhanden. Diese Klasse führt ein internes Zustandsmodell mit den beiden Eigenschaften *lebendig* (alive) und *ausgeführt* (done) ein. Ein

neues `AbstractUndoableEdit`-Objekt hat zunächst den Zustand lebendig und ausgeführt. Durch Aufruf von `die()` ist das Objekt nicht mehr lebendig und durch Aufruf von `undo()` nicht mehr ausgeführt. Durch `redo()` kann der Zustand *ausgeführt* wieder erreicht werden, soweit das Objekt vorher lebendig und nicht ausgeführt war. Dieses grundlegende Zustandsmodell kann von Unterklassen ausgenutzt werden, indem sie in ihrer `undo()`- bzw. `redo()`-Methode zunächst diese Methoden der Oberklasse aufrufen (`super.undo()` bzw. `super.redo()`). Dies funktioniert wie folgt:
In der Standardimplementierung liefert `canUndo()` *wahr*, wenn das `UndoableEdit`-Objekt lebendig und ausgeführt ist, `canRedo()` gibt *wahr* zurück, wenn es lebendig und nicht ausgeführt ist. Die Standardimplementierungen der Methoden `undo()` und `redo()`, die von einer Unterklasse von `AbstractUndoableEdit` aufgerufen werden, führen `canUndo()` bzw. `canRedo()` aus und werfen eine Ausnahme (`CannotUndo/RedoException`), wenn diese Methoden nicht *wahr* zurückliefern.

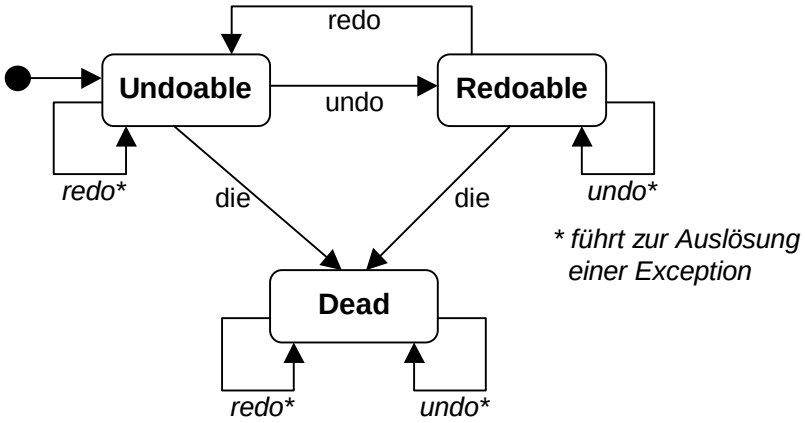

Abbildung 3.3: Zustandsdiagramm der Klasse UndoableEdit (aus: [Eck98])

Die Klasse CompoundEdit

Mit der Klasse `CompoundEdit`, einer Unterklasse von `AbstractUndoableEdit`, ist es möglich, mehrere kleine Änderungen zu einer einzigen großen zusammenzufassen. Ein typisches Anwendungsbeispiel hierfür ist ein Textverarbeitungsprogramm: Auch wenn die Eingabe jedes einzelnen Buchstabens im Prinzip ein eigener Befehl ist, wird, wenn der Benutzer die Undo-Funktion ausführt, meist das komplette zuletzt geschriebene Wort oder die komplette zuletzt geschriebene Zeile gelöscht.

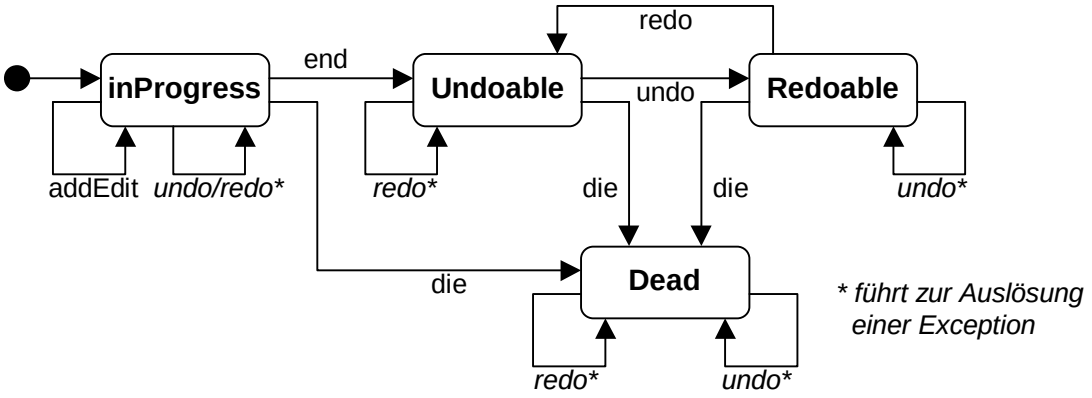

Abbildung 3.4: Zustandsdiagramm der Klasse CompoundEdit (aus: [Eck98])

Ein `CompoundEdit`-Objekt, das neu erzeugt wurde, hat zunächst den internen Zustand *in Bearbeitung* (in Progress). Es kann solange Objekte vom Typ `UndoableEdit` mit der Methode `addEdit()` aufnehmen bis die Methode `end()` aufgerufen wird. (`AddEdit()` gehört auch zur `UndoableEdit`-Schnittstelle, ist aber in der Klasse `AbstractUndoableEdit` ohne Funktion.) Dann wechselt der interne Zustand zu *ausgeführt* und das `CompoundEdit`-Objekt als ganzes kann nun rückgängig gemacht und wiederhergestellt werden. Dies geschieht indem das `CompoundEdit`-Objekt die `undo()`- bzw. `redo()`-Methode jeder einzelnen der zugefügten `UndoableEdit`-Objekte aufruft. Nach dem Aufruf von `end()` ist es nachträglich nicht mehr möglich, dem `CompoundEdit`-Objekt weitere `UndoableEdit`-Objekte hinzuzufügen.

Die UndoableEditEvent-Klasse

Bei dieser Klasse handelt es sich um eine Ereignisklasse, die von Undo-unterstützenden Komponenten benutzt wird. Die Komponenten können mit Hilfe dieser Klasse interessierte Beobachter (Listener) benachrichtigen, daß eine Änderung, die rückgängig gemacht werden kann, ausgeführt wurde.

Die UndoableEditListener-Schnittstelle

Dies ist die Schnittstelle, die Klassen implementieren müssen, wenn sie Benachrichtigungen über rückgängigmachbare Änderungen (`UndoableEditEvents`) empfangen wollen. Die einzige Methode der Schnittstelle ist `undoableEditHappend()`. Sie wird aufgerufen, wenn eine Operation auf einem Objekt, welches Undo unterstützt, ausgeführt wurde.

Die UndoManager-Klasse

Die `UndoManager`-Klasse ist die eigentliche Kernklasse des Swing-Undo-Pakets. Mit ihr ist es möglich, eine Befehlsgeschichte mit allen zurückliegenden Änderungen aufzubauen und ausgeführte Befehle einzeln nacheinander zurückzunehmen oder wiederherzustellen. Die `UndoManager`-Klasse ist eine Unterklasse von `CompoundEdit` und implementiert das `UndoableEditListener`-Interface. Der `UndoManager` kann sich bei mehreren Komponenten, die Undo unterstützen, als Beobachter anmelden und ist somit die zentrale Stelle, an der alle Änderungen registriert werden. Damit ist es möglich, ein Undo/Redo-Menü für eine komplette Anwendung zu benutzen.
Obwohl der `UndoManager` eine Unterklasse von `CompoundEdit` ist, hat sich die Funktionsweise in einigen Teilen erheblich geändert. Der Aufruf von `undo()` und `redo()` hat nicht mehr Auswirkungen auf alle gespeicherten Änderungen wie beim `CompoundEdit` sondern nur noch auf die jeweils letzte bzw. nächste. Während man zu einem `CompoundEdit` Änderungen nur hinzufügen kann, solange der Status in Bearbeitung ist und erst nach Ende der Bearbeitung Undo und Redo möglich sind, sind diese Funktionen beim `UndoManager` jederzeit verfügbar. Nach dem Aufruf von `end()` ändert der `UndoManager` seine Funktionsweise, und an die Stelle des sequentiellen Undos tritt die vom `CompoundEdit` bekannte Undo-Funktion. Dieses auf den ersten Blick etwas ungewöhnliche Verhalten soll dazu dienen, die Undo- und Redo-Funktionen sowohl für kleine Arbeitsschritte als auch für komplette Arbeitsvorgänge einsetzen zu können. Ein Anwendungsbeispiel hierfür ist ein Tabellenkalkulationsprogramm: Bei der Bearbeitung der Formel einer Zelle können mit `undo()` einzelne Teile der Formel wieder gelöscht werden. Nachdem der Benutzer die komplette Formel bestätigt hat, wird der Vorgang der Formelerstellung als ein einziger Schritt betrachtet, der nur noch komplett gelöscht werden kann. Abbildung 3.5 zeigt das komplette Zustandsdiagramm des Undo-

Managers. Auf eine genaue Beschreibung seines Zustandsmodells wird hier verzichtet. Sie kann bei Eckstein nachgelesen werden (vgl. [Eck98]).

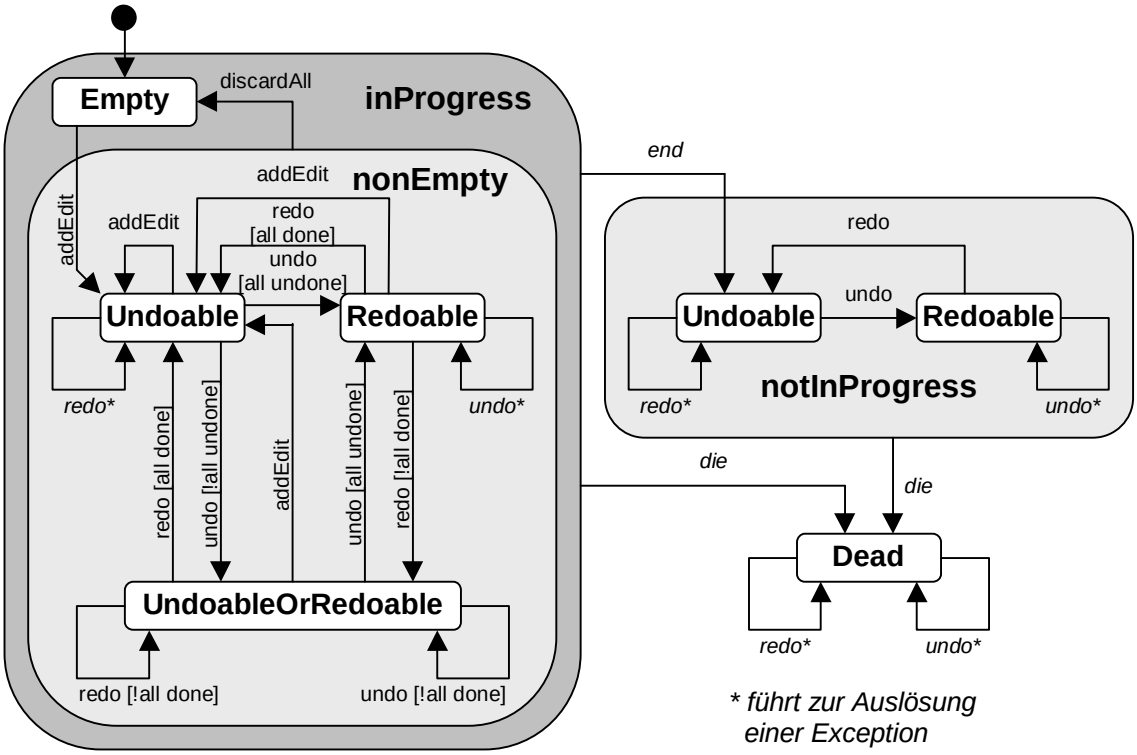

Abbildung 3.5: Zustandsdiagramm der Klasse UndoManager (aus: [Eck98])

Der `UndoManger` kann nur eine begrenzte Anzahl von Änderungen, die durch `undo()` oder `redo()` wiederhergestellt werden können, speichern. Diese Anzahl ist standardmäßig auf einhundert eingestellt, kann aber vergrößert oder verkleinert werden. Ein Sonderfall tritt auf, wenn das Limit auf eins beschränkt wird: Dann kann für die Darstellung an der Oberfläche die Methode `getUndoOrRedoPresentationName()` benutzt werden. Je nachdem ob der letzte Arbeitsschritt zurückgenommen wurde oder nicht, wird die Beschreibung von Undo oder Redo zurückgegeben. Entsprechend wird mit `undoOrRedo()` je nach aktuellem Zustand `undo()` oder `redo()` ausgeführt und mit `canUndoOrRedo()` abgefragt, ob dies möglich ist. Mit Hilfe dieser drei Methoden läßt sich eine Undo-Funktion nach dem Single-Undo-Modell, wie sie in Kapitel 2.3.1 beschrieben ist, realisieren.
Folgende Quellcodeauszüge zeigen ein kleines Programmbeispiel, dessen Undo-Funktion mit Hilfe der `UndoManager`-Klasse realisiert worden ist. Es sind nur die für die Undo-Funktion relevanten Stellen dargestellt. Die Anwendung erzeugt zwei An-Aus-Schalter, die durch Undo in eine früheren Zustand zurückversetzt werden können. Die dazugehörige Redo-Funktion würde entsprechend aussehen, ist aber aus Vereinfachungsgründen weggelassen worden.

```
// eine Beispielanwendung, die die Benutzung des UndoManagers demonstriert
public class UndoableExample extends JFrame
                             implements UndoableEditListener

{
  // erstellt das Hauptfenster und dessen Inhalt (zum Teil gekürzt)
  public UndoableExample()
  {
```

```
    // erstellt die Knöpfe und fügt zu jedem einen Beobachter hinzu
    UndoableToggleButton button1 = new UndoableToggleButton("One");
    button1.addUndoableEditListener(this);
    UndoableToggleButton button1 = new UndoableToggleButton("Two");
    button2.addUndoableEditListener(this);

    // erstellt den Undo-Knopf (zunächst nicht funktionsfähig)
    undoButton = new JButton("Undo");
    undoButton.setEnabled(false);

    // fügt dem Undo-Knopf einen Beobachter hinzu und versucht nach seiner
    // Auslösung undo() aufzurufen;
    // schaltet Undo-Knopf in richtigen Zustand (an/aus)
    undoButton.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent event)
      {
        try { manager.undo(); }
        catch (CannotUndoException e) {}
        finally { updateButtons(); }
      }
    });
  }

  // fügt dem UndoManager generierte UndoableEditEvents hinzu,
  // aktualisiert Beschriftung und Zustand der Knöpfe
  public void undoableEditHappend(UndoableEditEvent event)
  {
    manager.addEdit(event.getEdit());
    updateButtons();
  }

  // setzt oder ändert Beschriftung und Zustand der Knöpfe
  protected void updateButtons()
  {
    undoButton.setText(manager.getUndoPresentationName());
    undoBitton.setEnabled(manager.canUndo());
  }

  // Undo-Manager und Undo-Knopf
  private UndoManager manager = new UndoManager();
  private JButton undoButton;
}

// Beispielknopf UndoableToggleButton, für den Undo implementiert wird
// er ist eine Erweiterung des normalen ToggleButtons
public class UndoableToggleButton extends JToggleButton
{
  // um das Beispiel einfach zu halten, gibt es nur einen Konstruktor
  public UndoableToggleButton(String text)
  {
    super(text)
  }

  // setzt den UndoableEditListener
  public void addUndoableEditListener(UndoableEditListener uel)
  {
    listener = uel;
  }
```

```
  // entfernt den UndoableEditListener
  public void removeUndoableEditListener(UndoableEditListener uel)
  {
    listener = null;
  }

  // löst zunächst das ActionEvent und dann ein neues UndoableEditEvent aus
  protected void fireActionPerformed(ActionEvent ae)
  {
    super.fireActionPerformed(ae);
    if (listener != null)
    {
      listener.undoableEditHappend(new UndoableEditEvent
            (this, new UndoableToggleEdit(this)));
    }
  }
  private UndoableEditListener listener;
}

// diese Klasse beinhaltet die Implementierung der Undo-Funktion des
// ToggleButtons
public class UndoableToggleEdit extends AbstractUndoableEdit
{
  // erzeugt eine neue Änderung eines Knopfes, der gerade gedrückt wurde
  public UndoableToggleEdit(JToggleButton button)
  {
    this.button = button;
    selected = button.isSelected();
  }

  // liefert den Namen dieser Änderung
  public String getPresentationName()
  {
    return "Toggle" + button.getText() + " " + (selected ? "on" : "off");
  }

  // die Undo-Funktion versetzt den Knopf in den entgegengesetzten Zustand
  public void undo() throws CannotUndoException
  {
    super.undo():
    button.setSelected(!selected);
  }

  // der ToggleButton und sein Zustand
  private JToggleButton button;
  private boolean selected;
}
```

Neben den bisher vorgestellten Klassen gibt es im `javax.swing.undo`-Paket noch zwei weitere Klassen, die bei der Entwicklung von Undo- und Redo-Funktionen hilfreich sein können. Die Klasse `StateEdit` und das dazugehörige Interface `StateEditable` erlauben es, daß ein externes Objekt selbst seinen Zustand vor und nach einer Serie von Änderungen sichert und die Verantwortung somit nicht mehr beim `UndoableEdit-Objekt` liegt.
`UndoableEditSupport` ist eine Hilfsklasse für Klassen, die Undo unterstützen. Sie stellt Methoden bereit, die den Umgang mit Beobachtern und Ereignissen erleichtern.

3.3.3 Bewertung des Swing-Undo-Konzepts

Das Swing-Undo-Paket bietet eine umfassende Unterstützung für die Entwicklung von Java-Programmen mit Undo- und Redo-Funktion. Allerdings basiert diese Unterstützung auf einer recht komplexen, wenn auch strukturiert aufgebauten Klassenhierarchie. Daher ist ein gewisser Einarbeitungsaufwand notwendig, um die einzelnen Klassen des Pakets korrekt zu nutzen.
Anwendungen, die auf dem `javax.swing.undo`-Paket basieren, müssen sich an dessen Struktur anpassen. Der Mehraufwand an Entwicklungsarbeit gegenüber einer Anwendung ohne Undo-Unterstützung ist, wie auch das zuvor aufgeführte Beispiel zeigt, insbesondere bei kleineren Anwendungen erheblich.
Obwohl das Paket ein Teil von Swing ist, benutzt es keine weiteren Swing-Komponenten. Es läßt sich somit komplett unabhängig von den graphischen, wie auch allen anderen Swing-Komponenten einsetzen. Damit ist es auch möglich, das Paket mit einer anderen graphischen Bibliothek als Swing zu benutzen. Trotzdem sind die entstehenden Undo-Funktionen immer eng mit der graphischen Oberfläche der Anwendung verbunden. Direkt an den Komponenten der Oberfläche „lauschen“ die Beobachter, die diejenigen Methoden auslösen, die für die Speicherung der für Undo und Redo notwendigen Daten zuständig sind. Eine Trennung von Interaktion und Funktion, wie sie das WAM-Modell vorsieht (vgl. [Zül98]), ist somit nicht möglich.
Zusammenfassend betrachtet bietet das Swing-Undo-Paket eine gute Möglichkeit, Java-Anwendungen mit Undo-Unterstützung zu entwickeln. Voraussetzung dafür ist allerdings, daß sich die Architektur der Anwendung an den Vorgaben der Architektur des `javax.swing.undo`-Pakets orientiert. Um bestehende Anwendungen um eine Undo-Funktion zu erweitern, ist dieses Paket dagegen weniger geeignet, da die Anpassungen der Architektur meist eine komplette Neuprogrammierung notwendig machen.

3.4 Realisierung einer Undo-Funktion am Beispiel von JHotDraw

JHotDraw ist ein in Java geschriebenes Rahmenwerk zur Entwicklung von Zeichenprogrammen für kleine einfache Graphiken. Seine Ursprünge gehen auf das Mitte der achtziger Jahre von Beck und Cunningham in Smalltalk geschriebene Programm HotDraw zurück (vgl. [Joh92]). Dessen Ideen wiederum basieren auf dem von Vlissides entwickelten Unidraw-Rahmenwerk (vgl. [Vli90]). Gamma und Eggenschwiler portierten HotDraw nach Java und erweiterten es zu einem individuell anpaßbarem Rahmenwerk. Gleichzeitig veröffentlichten sie den Quellcode und stellten diesen frei für eigene Verwendungen zur Verfügung. Mittlerweile wird JHotDraw als Open-Source-Projekt von einer größeren Anzahl Interessierter weiterentwickelt (siehe [Jhd02]). Im Rahmen dieser Entwicklung erhielt JHotDraw mit der Version 5.3, die im Januar 2002 fertiggestellt wurde, eine Undo-/Redo-Funktion.
Bei der Entwicklung von JHotDraw wurde eine Vielzahl von Entwurfsmustern verwendet. (Die Anwendung gilt in dieser Hinsicht als wegweisend für die weitere Verbreitung von Entwurfsmustern.) Unter anderem wird für alle graphischen Aktionen an der Oberfläche das Befehlsmuster (siehe Kapitel 3.2.1) benutzt. Dies war einer der Gründe, der es möglich machte, JHotDraw ohne größere Änderungen gegenüber der Vorversion so zu erweitern, daß das Zurücknehmen und erneute Ausführen von Aktionen unterstützt wird. Abbildung 3.6 zeigt eine Anwendung, die mit dem JHotDraw-Rahmenwerk entwickelt wurde. Sie liegt dem Rahmenwerk als Beispiel bei. Es handelt sich um ein kleines Zeichenprogramm, mit dem verschiedene graphische Elemente erzeugt und miteinander verbunden werden können.

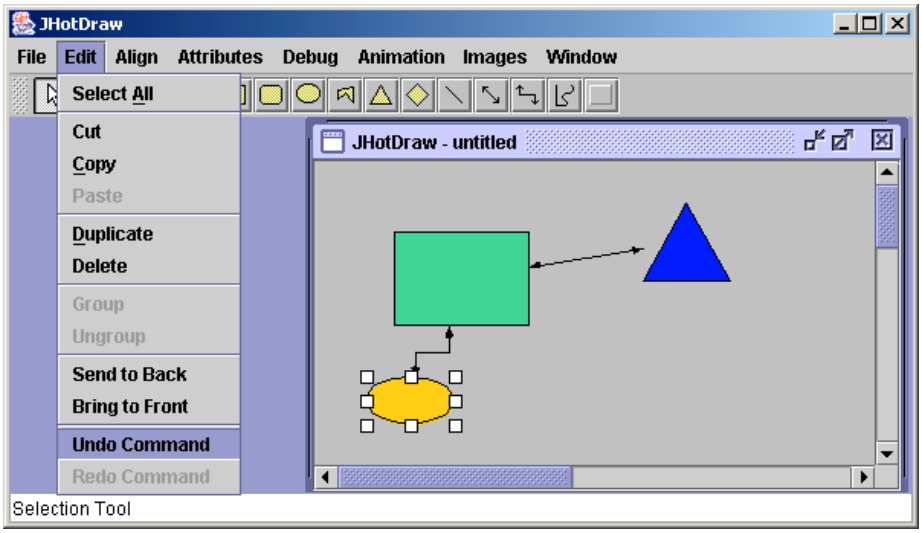

Abbildung 3.6: Beispielanwendung aus dem JHotDraw-Rahmenwerk

3.4.1 Aufbau der Undo-Funktion in JHotDraw

Wie bereits erwähnt, basiert JHotDraw auf dem Befehlsmuster. Das heißt, daß für alle dem Benutzer zur Verfügung stehenden Aktionen eigene Klassen existieren, die den jeweiligen Befehl kapseln. Jede dieser Befehlsklassen muß die Schnittstelle `Command` implementieren, in der die für alle Befehlsobjekte notwendigen Methoden festgelegt sind. Dies sind die Methoden, um die Ausführung des eigentlichen Befehls zu starten (`execute()`), um auf den Namen des Befehlsobjekts zuzugreifen (`name()`), sowie eine JHotDraw-spezifische Methode, die den Zugriff auf die Anwendung, zu der dieser Befehl gehört, erlaubt (`getDrawingEditor()`). Zusätzlich sind in der Befehlsschnittstelle Methoden vorgegeben, die es erlauben, eine Undo-Aktivität (siehe unten) zu setzen oder auszulesen (`get/setUndoActivity()`).
Eine Implementierung und zugleich weitere Spezialisierung der allgemeinen Befehlsschnittstelle erfolgt in den Klassen `AbstractCommand` und `UndoableCommand`. Die Klasse `AbstractCommand` stellt ein Grundgerüst für alle weiteren in JHotDraw verwendeten Befehle bereit. Neben den durch die implementierte Schnittstelle vorgegebenen Methoden beinhaltet diese Klasse hauptsächlich Methoden, die Zugriff und Verwaltung der einzelnen am Befehl beteiligten Oberflächenelemente regeln. Außerdem sind in dieser Klasse diverse Ereignisse (Events) definiert, die das Zusammenspiel zwischen Befehl und Anwendung regeln. Die Anwendung kann sich an diesen Ereignissen (z.B. „Befehl ausgeführt“) registrieren und entsprechend darauf reagieren.
Von der Klasse `AbstractCommand` wiederum werden einerseits alle normalen Befehlsklassen der Anwendung abgeleitet (zum Beispiel die Befehle, mit denen graphische Objekte ausgeschnitten oder eingefügt werden können). Andererseits sind auch die speziellen Undo- und Redo-Befehle (`Undo/RedoCommand`) Unterklassen von dieser. Undo- bzw. Redo-Befehle sind diejenigen, die ausgeführt werden, wenn der Benutzer an der Anwendung den Undo- bzw. den Redo-Knopf betätigt.
Nicht zu verwechseln mit dem `UndoCommand` ist die Klasse `UndoableCommand`, die ebenfalls die allgemeine Befehlsschnittstelle implementiert. Diese Klasse ist ein Dekorierer für alle normalen Befehle. (Das Dekorierer-Muster ist ein Entwurfsmuster, mit dem eine Klasse um zusätzliche Methoden ergänzt werden kann. Die Schnittstelle der dekorierten Klasse bleibt dabei unverändert, Vererbung wird nicht eingesetzt, siehe [Gam96].) Bei der Konstruktion von Objekten dieser Klasse wird jeweils das normale Befehlsobjekt, das um zusätzliche

Funktionalität erweitert (dekoriert) werden soll, als Parameter übergeben. Alle Methoden die am Dekorierer (`UndoableCommand`) aufgerufen werden, werden unverändert am dekorierten Befehl (normalen Befehl) ausgeführt. Wenn die `execute()`-Methode aufgerufen wird, sichert der Dekorierer mit Hilfe des Undo-Managers zusätzlich die Informationen, die zur Wiederherstellung des aktuellen Zustands mit der Undo-Funktion notwendig sind. Dazu fragt `UndoableCommand` am dekorierten Befehl die `UndoActivity` mit Hilfe der Methode `getUndoActivity()` ab und speichert sie im Undo-Manager. Damit dies bei jedem Befehl möglich ist, wurde, wie bereits zuvor erwähnt, die allgemeine `Command`-Schnittstelle um eine Zugriffs- und eine Setzmethode für die `UndoActivity` erweitert. Diese Erweiterung der Schnittstelle macht es notwenig, jeden Befehl aus einer älteren Version von JHotDraw um die zwei genannten Methoden zu erweitern. Befehle, die kein Undo und Redo unterstützen, müssen diese Methode leer implementieren.

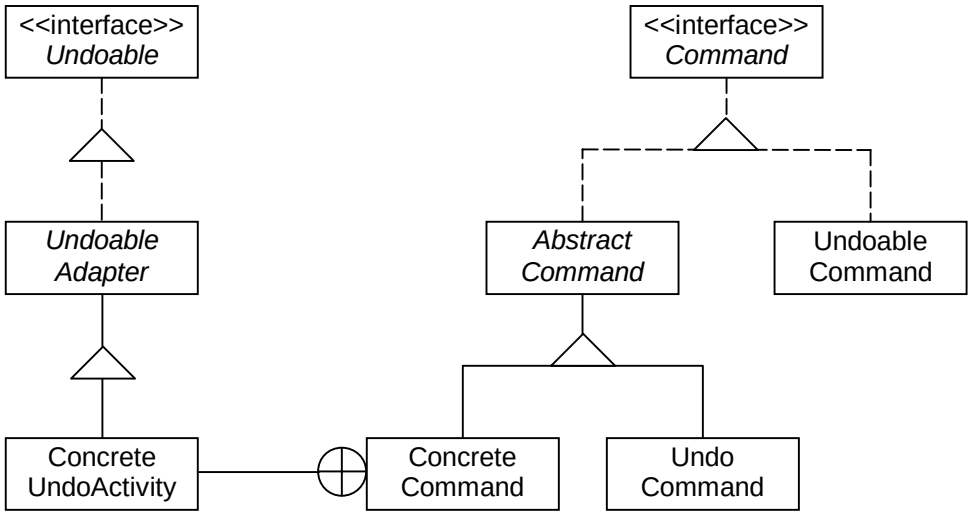

Abbildung 3.7: Klassenstruktur der Befehls- und Undo-Aktivitätsklassen in JHotDraw

Die `UndoActivity` ist die Klasse, die in JHotDraw verwendet wird, um alle für die Undo- und Redo-Funktion eines Befehls notwendigen Daten zu sichern. Außerdem sind in ihr die Methoden implementiert, mit denen die Auswirkungen eines Befehls rückgängig gemacht werden können (`undo()/redo()`). Sie wird als statische innere Klasse der zu ihr gehörenden Befehlsklasse realisiert (siehe Abbildung 3.7[1]).
Die von allen Undo-Aktivitäten zu implementierenden Methoden sind in der Schnittstelle `Undoable` festgelegt. Neben den eigentlichen Undo- und Redo Methoden (`undo()/redo()`, `isUndo(Redo)able()` und `setUndo(Redo)able()`) sind Methoden vorhanden, mit denen alle Oberflächenelemente, auf die ein Befehl Auswirkungen hat, gesichert oder zurückgelesen werden können (`get/setAffectedFigures()`). Diese gesicherten Elemente benutzen die `undo()`- und die `redo()`-Methode, um den Zustand vor bzw. nach Ausführung des Befehls wiederherzustellen. Auch für die Schnittstelle `Undoable` gibt es eine vorgegebene Standardimplementierung: Die Klasse `UndoableAdapter` dient als Superklasse für alle Undo-Aktivitäten.
Als zentrale Verwaltung für alle gespeicherten Undo-Aktivitäten dient der Undo-Manager. Er benutzt intern zwei als Kellerspeicher (Stack) aufgebaute Listen – eine, in die zunächst alle

[1] Das zur Darstellung dieser Beziehung in Abb. 3.7 und Abb. 3.8 verwendete Symbol ist seit Version 1.4 Bestandteil der UML-Definition und deutet eine Verankerung der ersten Klasse in der zweiten an (siehe [Uml01]).

Undo-Aktivitäten gespeichert werden, wenn ihr zugehöriger Befehl ausgeführt wurde (undoStack) und eine, in die diese Undo-Aktivitäten verschoben werden können, falls auf ihnen ein Undo ausgeführt wurde (redoStack). Der Undo-Manager speichert nur Undo-Aktivitäten, die Undo unterstützen. Im negativen Fall (`undoActivity.isUndoable() == false`) wird die Undo-Liste geleert und ein weiteres Undo ist nicht mehr möglich. Das Gleiche gilt für Redo. Um die korrekte Abstimmung der beiden Listen müssen sich die den Undo-Manager benutzenden Klassen selbst kümmern. Wenn ein neuer Befehl ausgeführt wird, ist es erforderlich, die Redo-Liste zu leeren, um Inkonsistenzen zu vermeiden. Dies geschieht zusammen mit der Speicherung der Undo-Aktivität in der Klasse `UndoableCommand`. Nachdem ein Befehl mittels Undo rückgängig gemacht wurde, muß die zugehörige Aktivität in die Redo-Liste verschoben werden, um seine Wiederherstellung zu erlauben. In JHotDraw ist hierfür der Undo-Befehl zuständig. Abbildung 3.8 stellt die Benutzungsbeziehungen der am Undo-Konzept von JHotDraw beteiligten Klassen dar.

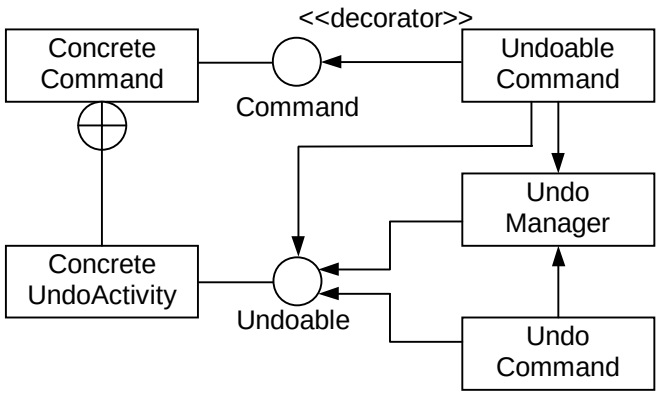

Abbildung 3.8: Benutzungsbeziehungen der an der Undo-Funktion von JHotDraw beteiligten Klassen

3.4.2 Funktionsweise der Undo-Funktion in JHotDraw

Sicherung eines Befehls

Viele der in JHotDraw vorhandenen Befehle stehen dem Benutzer in Form von Menüeinträgen, die beim Anklicken ausgeführt werden, direkt zur Verfügung. Benutzt wird dafür eine spezielle Befehlsmenüklasse (`CommandMenue`) die eine Erweiterung der normalen Swing-Menüklasse (`JMenu`) ist. Befehle, die einem Befehlsmenü hinzugefügt werden, erhalten automatisch einen Befehlsbeobachter (`CommandListener`), der die Ausführen-Methode dieses Befehls aufruft, sobald der Menüpunkt aktiviert wird. Für alle Befehle, die Undo unterstützen sollen, wird dabei ein Objekt der Klasse `UndoableCommand` erzeugt, das wiederum als Parameter für seinen Konstruktor das eigentliche Befehlsobjekt bekommt. (Es wird gewissermaßen eingepackt.) Das erstgenannte Objekt wird dann zum Befehlsmenü hinzugefügt. Der Programmcode, um z.B. den Gruppierungsbefehl (ein Befehl in JHotDraw, der mehrere graphische Elemente zusammenfaßt) an das Editiermenü zu hängen, sieht wie folgt aus:

```
CommandMenu menu = new CommandMenu("Edit");
menu.add(new UndoableCommand(new GroupCommand("Group", this)));
```

Der normale Befehl (hier: `GroupCommand`) wird im `UndoableCommand` gespeichert. Alle Methodenaufrufe am `UndoableCommand` werden unverändert an diesem gespeicherten Befehl ausgeführt. Wird die Ausführen-Methode eines `UndoableCommand` aufgerufen, so wird, nachdem der Aufruf weitergeleitet wurde, die Undo-Aktivität des speziellen Befehls abgefragt (`getUndoActivity()`) und im Undo-Manager gespeichert (`pushUndo (undoableCom-`

mand)). Diese Undo-Aktivität erlaubt es, den Befehl später rückgängig zu machen. Dazu legt sie Kopien aller Oberflächenelemente an, die durch den zugehörigen Befehl verändert werden. Wie jeder einzelne Befehl genau rückgängig gemacht wird, variiert von Befehl zu Befehl und muß deshalb für jede Klasse neu implementiert werden.

Rückgängigmachen eines Befehls

Ausgelöst wird die Aktion, einen Befehl rückgängig zu machen, dadurch, daß aus dem Befehlsmenü am Undo-Befehl die Ausführen-Methode aufgerufen wird. In dieser Methode wird am Undo-Manager die letzte Undo-Aktivität der Undo-Liste abgefragt (`undoManager.popUndo()`) und an dieser die Undo-Methode aufgerufen. Sie sorgt dafür, daß alle Oberflächenelemente in den Zustand zurückversetzt werden, den sie vor der Ausführung des Befehls hatten. Dazu werden die am Befehl beteiligten Elemente abgefragt (`getAffectedFigures()`) und anschließend in ihrem alten Zustand rekonstruiert. Nach der Ausführung der Undo-Methode wird, sofern der Befehl Redo-fähig ist, seine Undo-Aktivität in der Redo-Liste gespeichert. Diese Undo-Aktivität wird wieder abgefragt, wenn am Befehlsmenü der Redo-Befehl aufgerufen wird. Dies führt zur Ausführung der Redo-Methode an der gespeicherten Undo-Aktivität und somit zur Rekonstruktion des Anwendungszustands vor Ausführung des Undo-Befehls.

Handles

Neben dem Befehlsmuster werden in JHotDraw auch sogenannte *Handles* verwendet, um Änderungen an graphischen Objekten durchzuführen. Es handelt sich hierbei um Objekte, die es erlauben, direkt auf Oberflächenelemente zuzugreifen, an ihnen Änderungen vorzunehmen, sie zu überwachen oder auftretende Zustandsänderungen (z.B. Maus geklickt) zu verfolgen. Das Konzept der Handles ist ähnlich wie das der in JWAM verwendeten Manipulatoren (siehe Kapitel 4.1).
Die Undo-Funktion für Handeles entspricht der von normalen Befehlen: Ein `undoableHandle` kapselt ein normales Handle und reicht alle Methodenaufrufe an dieses weiter. Die Sicherung des aktuellen Zustands erfolgt, nachdem das Handle seine Aktion abgeschlossen hat (`invokeEnd()`). Dazu wird, wie bei einem normalen Befehl, eine neue Undo-Aktivität erzeugt und anschließend im Undo-Manager abgespeichert.

3.4.3 Bewertung des Undo-Konzepts von JHotDraw

Da JHotDraw bereits vor der Implementierung der Undo-Funktion vollständig auf dem Befehlsmuster basierte, konnten seine Entwickler diese Erweiterung relativ einfach einbauen. Die dabei entstandene Architektur ergänzt die bisherige um neue Klassen, verändert aber nicht die grundlegende Struktur. Allerdings wurden innerhalb der bestehenden Klassen und Schnittstellen auch Ergänzungen vorgenommen (es wurden die Methoden `get/setUndoActivity()` in `Command` eingeführt), die es verhindern, alte Befehlsklassen unverändert weiterzubenutzen, auch wenn für sie kein Undo/Redo implementiert werden soll. In Anwendungen, die mit einer älteren Version des JHotDraw-Rahmenwerks entwickelt wurden, ist es erforderlich, alle Befehlsklassen an die neue Befehlsschnittstelle anzupassen, sobald für einen einzigen Befehl Undo bereitgestellt werden soll.
Zusammenfassend betrachtet eignet sich das in JHotDraw verwendete Undo-Konzept gut, um in neu zu entwickelnde Anwendungen eine Undo-Funktion einzubauen. Dafür ist es allerdings notwendig, daß sich die Architektur des neuen Programms an der von JHotDraw orientiert. Bestehende Anwendungen mit abweichendem Aufbau lassen sich dagegen nur mit großen Anpassungen um eine Undo-Funktion ergänzen.

Ein Nachteil des Undo-Konzepts von JHotDraw liegt darin, daß bereits in grundlegenden Klassen und Schnittstellen JHotDraw-spezifischer Programmcode verwendet wird. Beispielsweise sind in der Schnittstelle `Undoable`, die das Grundgerüst für alle Undo-Aktivitäten vorgibt, Methoden definiert, die nur für graphische Elemente von JHotDraw Sinn machen (z.B. `setAffectedFigures()`) und ohne die die Undo-Funktion nicht arbeiten kann. Deswegen müßten nahezu alle Klassen und Schnittstellen angepaßt werden, wenn das Undo-Konzept von JHotDraw auf eine andere Anwendung übertragen werden soll.

3.5 Zusammenfassung

Alle drei vorgestellten Lösungen zur Realisierung einer Undo-Funktion beruhen auf der gleichen Idee und sind sich daher recht ähnlich: Sie bauen alle auf dem Befehlsmuster auf. Der Grund dafür ist darin zu finden, daß mit keiner anderen Lösungsidee eine Undo-Funktion in ähnlich einfacher und universell verwendbarer Weise realisiert werden kann.
Die Unterschiede zwischen den einzelnen Beispielen liegen eher im Detail: Während Gamma et al. vorschlagen, jeweils das komplette Befehlsobjekt als Grundlage für die Wiederherstellung zu speichern (vgl. [Gam96]), werden dafür in Swing und JHotDraw jeweils Objekte einer eigenen Klasse verwendet (`UndoableEdit` bzw. `UndoActivity`). Da diese Klassen nicht die komplette Programmlogik des Befehls enthalten, sondern nur die für Undo und Redo benötigten Informationen, ist der Umfang der für einen Arbeitsschritt zu speichernden Daten wesentlich geringer.
Durch die Verwendung des Befehlsmusters steigt die Anzahl der Klassen einer Anwendung stark an, da für jeden Befehl eine eigene Klasse notwendig ist. Diese Anzahl verdoppelt sich noch einmal, wenn für jeden Befehl zusätzlich eine Klasse zur Speicherung der Undo-Informationen, wie die `UndoableEdit`-Klasse in Swing, notwendig ist. Bei JHotDraw wird dieses Problem dadurch umgangen, daß die `UndoActivity`-Klassen als innere Klassen des Befehls, zu dem sie gehören, implementiert werden. Das führt zu einer übersichtlicheren Klassenstruktur und macht zudem deutlich, daß die `UndoActivity`-Klassen immer genau einem Befehl zugeordnet werden. Für die Implementierung hat die Verwendung von inneren Klassen den Vorteil, daß direkt auf die Variablen der umgebenden Befehlsklasse zugegriffen werden kann, ohne ihre äußere Datenkapselung zu verletzen. Wäre die `UndoActivity`-Klasse eine normale Klasse, müßten für alle Variablen und Objekte, die gesichert und nach einem Undo wiederhergestellt werden sollen, Zugriffs- und Zuweisungsmethoden implementiert werden.
Bei den in Swing und JHotDraw integrierten Undo-Konzepten handelt es sich um umfassende Konzepte, die weit über das einfache Zurücknehmen eines Arbeitsschritts hinausgehen. Leider verlieren sie dadurch ihre allgemeine Einsetzbarkeit. Während das Undo-Paket aus Swing nur den Einsatz dieser Bibliothek voraussetzt, ist die Undo-Funktion aus JHotDraw auf den Einsatz in graphischen Anwendungen, die mit dem gleichnamigen Rahmenwerk erstellt wurden, begrenzt, sofern man nicht große Teile des Programmcodes der Undo-Funktion neu implementieren will.
Zur Erweiterung bestehender Anwendungen um eine Undo-Funktion sind beide Ansätze schlecht geeignet, da sie eine komplette Anpassung der Anwendung erforderlich machen. In Kapitel 5 wird deswegen ein Lösungsansatz vorgestellt, der es auch erlaubt, eine Anwendung nachträglich und nur in Teilen mit einer Undo-Funktion auszustatten.

4 Undo/Redo-Konzepte für manipulatorbasierte Werkzeuge

Nachdem im vorigen Kapitel die Lösungen zur Realisierung einer Undo-Funktion von mehreren bekannten Beispielprogrammen vorgestellt wurden, beschäftigen sich dieses und das nächste Kapitel mit allgemeinen Undo-Konzepten für JWAM-basierte Programme. Die Konzepte der zuvor analysierten Programme dienen dabei als Grundlage.
In diesem Kapitel wird ein Undo-Konzept vorgestellt, das auf eine spezielle Klasse von JWAM-Werkzeugen, auf die manipulatorbasierten Werkzeugen, abgestimmt ist. Werkzeuge, die dieser oder einer ähnlichen Struktur entsprechen, können mit dem vorgestellten Konzept relativ einfach um eine Undo-Funktion erweitert werden. Für die meisten anderen Werkzeuge muß dagegen zunächst die Architektur angepaßt werden, um dieses Konzept nutzen zu können. Da das Manipulatorkonzept (siehe Kapitel 4.1) relativ nahe mit dem Befehlsmuster verwandt ist, gilt dies nicht für befehlsmusterbasierte Werkzeuge: Auf diese Klasse von Werkzeugen kann das vorgestellte Konzept mit wenigen Anpassungen übertragen werden.
Es werden fünf Varianten einer auf diesem Konzept basierenden Undo-Funktion vorgestellt und auf ihre Vor- und Nachteile untersucht. Als Beispielanwendung dient dabei das mit Hilfe von Manipulatoren entwickelte „Graphical Modelling Tool“ (siehe Kapitel 4.2).

4.1 Manipulatoren

Manipulatoren sind kleine werkzeugartige Programmteile, die eine bestimmte Funktionalität einer Anwendung realisieren. Sie können nur zusammen mit einem ihnen übergeordneten Werkzeug eingesetzt werden. Die Manipulatoren und dieses Werkzeug bilden zusammen eine manipulatorbasierte Anwendung. Manipulatoren besitzen keine eigene Oberfläche, sondern greifen auf die des ihnen übergeordneten Werkzeugs zu.
Das Konzept der Manipulatoren ist dem der Befehlsobjekte aus dem Befehlsmuster (vgl. Kapitel 3.2.1) sehr ähnlich: Alle Manipulatoren einer Anwendung sind gleich aufgebaut. Sie besitzen, wie Befehlsobjekte, eine einheitliche Methode, durch die die vom jeweiligen Manipulator auszuführende Aktion gestartet wird. Diese Methode (`manipulate()`, sie entspricht der `execute`()-Methode im Befehlsmuster) ist in der Schnittstelle `Manipulator` festgelegt. Diese Schnittstelle muß von allen konkreten Manipulatoren mit der für die zugehörige Aktion notwendigen Funktionalität implementiert werden.
Mit diesem Aufbau ist es möglich, Anwendungen um zusätzliche Manipulatoren und damit um neue Funktionalität zu erweitern, ohne große Änderungen am Quellcode der übergeordneten Werkzeugklasse vornehmen zu müssen. Es muß lediglich im übergeordneten Werkzeug ein Exemplar des neuen Manipulators erzeugt werden. Die Aktion des neuen Manipulators kann genauso wie eine Aktion der bereits existierenden Manipulatoren aufgerufen werden.
Der Unterschied zwischen Befehlen und Manipulatoren liegt darin, daß Manipulatoren direkten Zugriff auf die Oberfläche des Werkzeugs, zu dem sie gehören, haben. Sie können so Oberflächenelemente ohne den Umweg über die Anwendung modifizieren. Im Gegensatz dazu werden mit Befehlsobjekten meist nur fachliche Aktionen ausgeführt.
Der Begriff *Manipulator* ist bisher nicht sehr verbreitet. In vielen Anwendungen (beispielsweise JHotDraw) werden Klassen, die den Aufbau eines Manipulators besitzen, als Befehle bezeichnet. Eine ausführliche Beschreibung des Manipulatorkonzepts findet sich in der Diplomarbeit von Simon Ditrich (siehe [Dit03]).

4.2 Das Graphical Modelling Tool

Das Graphical Modelling Tool ist ein kleines graphisches Werkzeug, mit dem einfache Zeichnungsobjekte und die Beziehungen zwischen ihnen dargestellt und verändert werden können. Die Klassen dieses Werkzeugs dienen als ein Grundgerüst, auf dessen Basis auch andere graphische Editoren entwickelt werden können. Sie bilden gewissermaßen, ähnlich wie JHotDraw, ein kleines Rahmenwerk zur Erstellung graphischer Anwendungen. Im Vergleich zu diesem ist die Anzahl der Klassen und somit die durch das Rahmenwerk bereitgestellte Funktionalität allerdings gering. Das Graphical Modelling Tool ist ein Bestandteil des JWAM-Rahmenwerks.
Das Graphical Modelling Tool kann zum Beispiel dafür benutzt werden, die Benutzungsbeziehungen aller Klassen einer Anwendung graphisch zu modellieren. Die Klassen werden dabei als Rechtecke und die Beziehungen zwischen ihnen als Pfeile (die sog. Konnektoren) dargestellt (siehe Abbildung 4.2). Der Benutzer des Werkzeugs kann die Darstellung beliebig verändern, indem er Rechtecke oder Pfeile verschiebt, neu erzeugt oder entfernt.

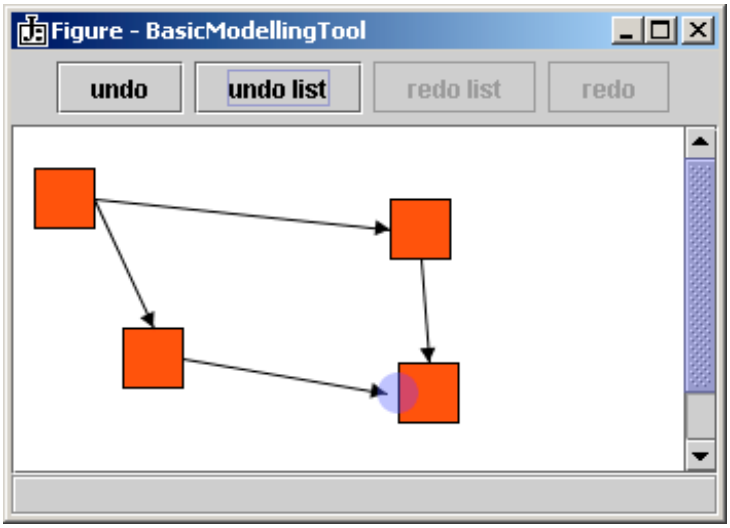

Abbildung 4.1: Das Graphical Modelling Tool

Aufgebaut ist das Graphical Modelling Tool folgendermaßen: In einem Werkzeugkern, der Klasse `toolBasicModellingTool`, ist die Grundfunktionalität zur Anzeige und zur Bearbeitung der verwendeten Materialien implementiert. Bei diesen Materialien handelt es sich um Exemplare der Klassen `Figure` und `FigureElement`. `figureElements` sind alle Zeichnungsobjekte, die im Graphical Modelling Tool bearbeitet werden können. Bereits Bestandteil des Werkzeugs sind die oben genannten Zeichnungsobjekte Rechteck und Pfeil. Wenn die Klassen des Graphical Modelling Tool als Basis für eigene Anwendungen verwendet werden, können auch weitere spezialisierte `figureElements` implementiert werden. Die Klasse `Figure` ist ein Behälter, in dem alle `figureElements` gespeichert werden, die an der Benutzungsoberfläche angezeigt werden.
Die Klasse `toolBasicModellingTool` dient außerdem als Registratur für alle in der Anwendung verwendeten Manipulatoren. Beim Start des Werkzeugs müssen an dieser Klasse alle Manipulatoren registriert werden, die dem Benutzer zur Verfügung stehen sollen. Das Graphical Modelling Tool benutzt insgesamt fünf verschiedene Manipulatoren, um Änderungen an der `Figur` durchzuführen:

- `AddElementManipulator`: erzeugt ein neues `figureElement` und fügt es zur `Figur` hinzu
- `RemoveElementManipulator`: löscht ein bestehendes `figureElement` aus der `Figur`
- `MoveElementManipulator`: verschiebt ein komplettes `figureElement`
- `MoveConnectorManipulator`: verschiebt den Anfang oder das Ende eines Konnektors; ermöglicht es, neue Verbindungen zu erstellen oder bestehende aufzulösen
- `MoveCoordinatorManipulator`: trifft die Entscheidung, welcher der beiden zuletzt genannten Manipulatoren jeweils ausgeführt wird

Alle Manipulatoren, die im Graphical Modelling Tool verwendet werden sollen, müssen von der gemeinsamen Oberklasse `Manipulator` abgeleitet werden. Sie ist, wie auch die Klasse `toolBasicModellingTool`, eine Unterklasse der JWAM-Klasse `ToolMonoImpl`. (`ToolMonoImpl` ist eine Rahmenwerksklasse, die ein Grundgerüst für die Konstruktion von Werkzeugen bereitstellt.) Somit haben alle Manipulatoren, bis auf die fehlende eigene Benutzungsoberfläche, die vollständige Funktionalität eines Werkzeugs.

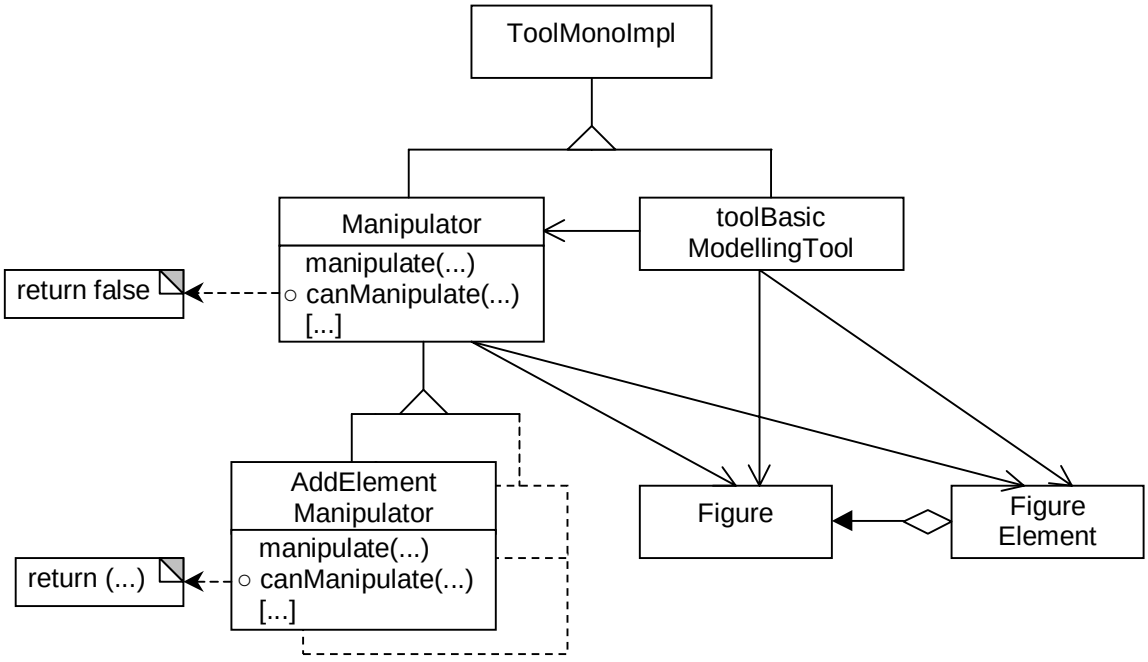

Abbildung 4.2: Architektur des Graphical Modelling Tools

Die Klasse `Manipulator` legt die von allen abgeleiteten Manipulatoren mit ihrer eigenen Funktionalität zu überschreibenden Methoden fest. Diese Methoden sind leer implementiert, so daß sie, wenn sie nicht überschrieben werden, keine Funktion besitzen. Die erste dieser Methoden ist die `manipulate()`-Methode. In dieser Methode muß die Funktionalität implementiert werden, die ausgeführt werden soll, wenn die Aktion des Manipulators aufgerufen werden soll. (Hier muß z.B. im `RemoveElementManipulator` der Programmcode stehen, mit dem ein `figureElement` aus der `figure` entfernt wird.) Die `manipulate()`-Methode entspricht damit der `execute()`-Methode in den Befehlen des Befehlsmusters. Eine Besonderheit dabei ist, daß die `manipulate()`-Methode mit vier festgelegten Parametern aufgerufen werden muß. Mit diesen Parametern werden Referenzen auf die vom Manipulator für die Bearbeitung notwendigen Materialien übergeben. Dadurch werden allerdings die Erweite-

rungsmöglichkeiten der im Graphical Modelling Tool eingesetzten Manipulator-Klassen eingeschränkt, da bei allen Manipulatoren die Anzahl und die Typen der Parameter mit den vorgegebenen übereinstimmen müssen.
Mit einer weiteren durch die Klasse `Manipulator` vorgegebenen Methode können alle Manipulatoren abgefragt werden, ob sie zum aktuellen Zeitpunkt ausführbar sind (`canManipulate()`). Auch diese Methode besitzt drei durch die Anwendung festgelegte Parameter. In Abbildung 4.2 sind alle zuvor vorgestellten Klassen und die Beziehungen zwischen ihnen in einem Klassendiagramm dargestellt.
Das Graphical Modellig Tool wurde als Beispielanwendung ausgewählt, da für die Entwicklung und Evaluierung der Undo-Funktionen zunächst eine möglichst einfach aufgebaute Anwendung benutzt werden sollte. Das Konzept der Manipulatoren, jeweils eine Aktion in einer einzelnen Klasse zu kapseln, erleichtert es zudem, die für diese Aktion relevante Funktionalität isoliert von dem Rest der Anwendung zu betrachten. Seiteneffekte, die die Implementierung der Undo-Funktionen erschweren, spielen somit in dieser Anwendung keine Rolle.

4.3 Undo/Redo-Konzepte für Manipulatoren

Mit Hilfe der im vorigen Unterkapitel vorgestellten Beispielanwendung, dem Graphical Modelling Tool, wurden verschiedene Implementierungsvarianten von Undo-Funktionen für manipulatorbasierte Anwendungen realisiert. Diese Lösungsvarianten lassen sich in zwei grobe Klassen unterteilen: Bei den ersten drei Implementierungen wird jeweils der komplette Manipulator, in dem die Informationen zur Wiederherstellung gespeichert sind, in der Befehlsgeschichte gesichert. Implementierungsvariante 4 und 5 nutzen eine eigene Klasse für die Speicherung der Wiederherstellungsinformationen.

4.3.1 Sicherung des vollständigen Manipulators

Alle in diesem Abschnitt vorgestellten Lösungsvarianten benutzen zunächst einmal das gleiche Grundkonzept zur Realisierung der Undo/Redo-Funktion. Es orientiert sich an dem in Kapitel 3.2.2 vorgestellten Undo-Konzept für Anwendungen, die das Befehlsmuster einsetzen. Jeder Manipulator wird um zwei zusätzliche Methoden erweitert, mit denen die zuletzt ausgeführte Aktion dieses Manipulators zurückgenommen oder das Ergebnis einer zuvor zurückgenommenen Aktion wiederhergestellt werden kann. Diese Methoden (`undo()`, `redo()`) werden mit einer leeren Implementierung in die allgemeine `Manipulator`-Klasse aufgenommen. Sie müssen von allen konkreten Manipulatoren mit den jeweils für die Rückname bzw. die Wiederherstellung dieser Aktion notwendigen Programmschritten überschrieben werden.
Um zu einem späteren Zeitpunkt die `undo()`- oder die `redo()`-Methode ausführen zu können, müssen in jedem Manipulator Informationen, die für die Wiederherstellung des Anwendungszustands vor oder nach der Ausführung der `manipulate()`-Methode notwendig sind, gespeichert werden. Die drei in diesem Abschnitt vorgestellten Lösungsvarianten unterscheiden sich im Umfang dieser Wiederherstellungsinformationen sowie der Art, wie damit Anwendungszustände rekonstruiert werden können.
Um eine mit Hilfe eines Manipulators realisierte Aktion ausführen zu können, muß zunächst ein neues Exemplar dieses Manipulators erzeugt werden. An ihm wird dann die `manipulate()`-Methode aufgerufen, die die Implementierung der gewünschten Aktion beinhaltet. Dabei werden zusätzlich die für Undo und Redo benötigten Informationen innerhalb des Manipulators abgespeichert. Anschließend wird der Manipulator selbst in einer globalen Liste

gespeichert. Bei dieser Liste handelt es sich um die Befehlsgeschichte, in der alle ausgeführten Manipulatoren gesichert werden.
Da die implementierte Undo/Redo-Funktion dem eingeschränkten linearen Undo-Modell entsprechen soll (siehe Kapitel 2.4.1), muß nur jeweils die zuletzt ausgeführte Aktion rückgängig gemacht oder die zuletzt rückgängig gemachte Aktion wiederhergestellt werden können. Dazu müssen die zu diesen Aktionen gehörenden Manipulatoren an der Befehlsgeschichte abgefragt werden und anschließend ihre `undo()`- bzw. `redo()`-Methode aufgerufen werden.
Die drei folgenden Lösungsvarianten bauen alle auf dem beschriebenen Konzept auf. Sie unterscheiden sich allerdings in den eingesetzten Wiederherstellungsstrategien (siehe Kapitel 3.1). Die erste Lösung benutzt die Strategie des inversen Befehls, die zweite die Strategie der vollständigen Sicherung und die dritte die Strategie der partiellen Sicherung.

Lösungsvariante 1: Undo mit Hilfe von „fachlichen Umkehrfunktionen“

Soweit dies möglich ist, bietet es sich an, die Umkehrung der in der `manipulate()`-Methode am Material aufgerufenen fachlichen Methode auszuführen, um eine Aktion rückgängig zu machen. Für die Redo-Funktion werden die gleichen Programmschritte wie in der `manipulate()`-Methode ausgeführt.
Beispielsweise ruft der `AddElementManipulator` an der `figure` die Methode `addElement()` auf, um ein zuvor erzeugtes neues Zeichnungsobjekt zu ihr hinzuzufügen. Als Information für die Undo-Funktion wird bei der Ausführung der `manipulate()`-Methode eine Referenz auf das hinzugefügte `figureElement` im Manipulator abgespeichert. Um diesen Vorgang rückgängig zu machen, wird in der `undo()`-Methode die `removeElement()`-Methode der `figure` mit dem `figureElement` als Parameter aufgerufen, die dieses Zeichnungsobjekt wieder aus der `figure` entfernt. Um die Aktion des `MoveElementManipulators`, bei dem der Benutzer mit Hilfe der Maus einzelne Elemente verschieben kann, rückgängig zu machen, wird `moveTo()` mit den Koordinaten des Ausgangspunkts aufgerufen. Auch diese Koordinaten müssen zuvor im Manipulator gesichert werden, damit sie später in der `undo()`-Methode benutzt werden können.
Der Vorteil dieser Lösungsvariante liegt darin, daß mit relativ wenig gesicherten Daten der ursprüngliche Zustand der Anwendung wiederhergestellt werden kann. Zudem werden nur bestehende fachliche Methoden des bearbeiteten Materials benutzt. Die fachliche Kapsel des Materials bleibt damit erhalten. Hier liegt aber auch das Hauptproblem, durch das diese Lösung nicht universell einsetzbar ist: Bei vielen Materialien kann nicht sichergestellt werden, daß jede ausgeführte fachliche Aktion durch eine entsprechende Aktion rückgängig gemacht werden kann. Oft ist es programmtechnisch gar nicht möglich, eine fachliche Aktion des Materials durch eine zweite fachliche Aktion rückgängig zu machen. Zum Beispiel kann eine Sortierfunktion eine Liste so umordnen, daß ihre Elemente eine bestimmte Reihenfolge bekommen. Die ursprüngliche Reihenfolge läßt sich aber durch keine fachliche Aktion wiederherstellen. Selbst wenn sich solche Aktionen leicht realisieren lassen, ist trotzdem nicht sichergestellt, daß sie tatsächlich existieren und zudem noch Bestandteil der öffentlichen Schnittstelle des Materials sind.

Lösungsvariante 2: Sicherung des kompletten Materials im Manipulator

Um auch für die Aktionen, für die keine Umkehrfunktionen existieren, eine Undo-Unterstützung realisieren zu können, benötigt man einen alternativen Lösungsweg: Statt die fachlichen Methoden des Materials einzusetzen, um die letzte Aktion rückgängig zu machen, wird das Material selbst (oder die Materialien, falls mehrere zugleich bearbeitet werden) vor der Ausführung der Aktion des Manipulators gesichert. Wenn die Undo-Funktion aufgerufen wird, wird das bearbeitete Material durch die zuvor erstellte Sicherung ersetzt. Für die Redo-

Methode wird dementsprechend das Material nach der Ausführung der Aktion des Manipulators abgespeichert. Nachdem Redo aufgerufen wird, ersetzt diese Version des Materials die zuvor bearbeitete. Die Sicherung der beiden für die Undo- und die Redo-Funktion benötigten Versionen des Materials erfolgt in dem zugehörigen Manipulator. Dieser Manipulator wird, wie in Lösungsvariante 1, nach seiner Ausführung in einer Manipulator-Liste abgespeichert.
Voraussetzung für den Einsatz dieser Lösung ist es, daß von dem bearbeiteten Material vollständige technische Kopien erzeugt werden können. Bei diesen Kopien muß es sich um tiefe Kopien handeln, bei denen neben dem Materialobjekt auch alle von diesem rekursiv referenzierten Objekte kopiert werden. Für die Undo/Redo-Funktion wird zu zwei verschiedenen Zeitpunkten eine Kopie dieser Art vom Material erzeugt. Einmal wird das Material vor und einmal nach Ausführung der `manipulate()`-Methode des Manipulators kopiert und im Manipulator gespeichert. Würde hier nur eine Referenz auf das Material oder eine flache Kopie gesichert werden, dann würden durch die weitere Bearbeitung des aktuellen Materials auch die Kopien verändert werden und somit deren Rekonstruktion im ursprünglichem Zustand nicht mehr möglich sein. In Abbildung 4.3 ist der Zustand des Programms dargestellt, nachdem die beiden Sicherungen der `Figure` erzeugt wurden. Während die aktuelle `Figure` mit Hilfe des Werkzeugs weiterhin bearbeitet werden kann, sind die beiden Sicherungen vor Veränderungen geschützt.

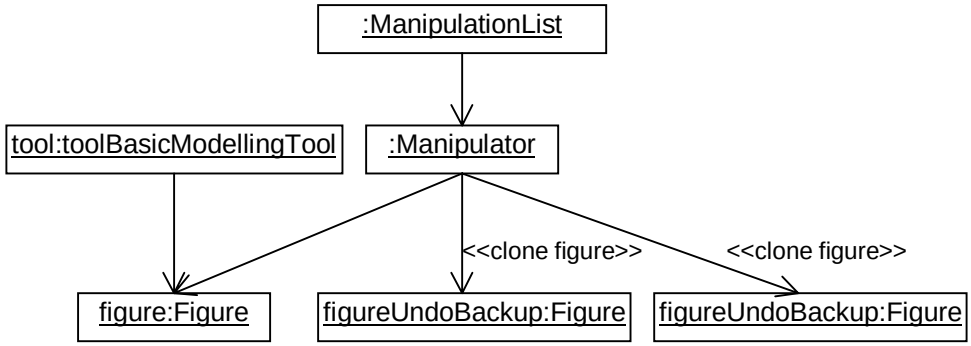

Abbildung 4.3: Objektstruktur nach der Ausführung einer Aktion mit Hilfe eines Manipulators

Zudem ist es notwendig, von jedem gesicherten Material eine neue tiefe Kopie zu erzeugen, wenn bei der Ausführung der `undo()`- oder `redo()`-Methode das aktuelle Material durch eine frühere Sicherung dieses Materials ersetzt werden soll. In diesem Fall darf das Werkzeug nur auf einer Kopie der Sicherung weiterarbeiten, da sonst die erste Sicherung bei der weiteren Bearbeitung verändert würde. Damit wäre es dann nicht mehr möglich, denselben Zustand ein zweites Mal zu rekonstruieren.
Von den Manipulatoren des Graphical Modelling Tools werden die Materialien `figure` und `figureElement` bearbeitet. Da sich in der `figure` Referenzen auf alle sichtbaren `figureElements` befinden, muß nur dieses Material für die Undo/Redo-Funktion gesichert werden. Nach Ausführung der `undo()`- oder `Redo()`-Methode eines Manipulators ersetzt eine Kopie der Sicherung der `figure` ihre ursprüngliche Version. Bis zu diesem Zeitpunkt hat allerdings nur der Manipulator, dessen `undo()`- bzw. `redo()`-Methode aufgerufen wurde, Zugriff auf die neue Version des Materials. Im übergeordneten Werkzeug (in diesem Fall `toolBasicModellingTool`), aus dem ursprünglich die `undo()`- oder `redo()`-Methoden am Manipulator aufgerufen wurden, muß ebenfalls die alte `figure` durch die neue Version ersetzt werden. Es würde sonst der nächste Manipulator mit einer `figure` als Parameter aufrufen, die nicht der aktuell in der Anwendung dargestellten entspricht. In Abbildung 4.4 ist der Anwendungszustand dargestellt, den die Anwendung besitzt, nachdem die Undo-Funktion ausgeführt wurde. Der Doppelpfeil deutet dabei die zu verschiebende Referenz an.

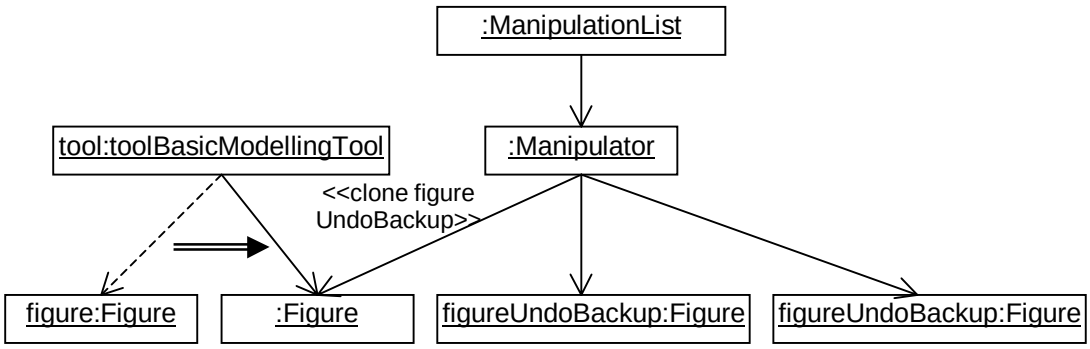

Abbildung 4.4: Objektstruktur nachdem die Undo-Funktion ausgeführt wurde

Da in Java die Verwendung von Zeigern nicht möglich ist, müssen hier andere Lösungen benutzt werden, damit das `toolBasicModellingTool` auf der neuen Version der `figure` weiterarbeitet. Wenn, wie in dieser Anwendung, nur ein Material verändert wird, kann seine neue Version als Rückgabewert der `undo()`- bzw. der `redo()`-Methode verwendet werden. Das `toolBasicModellingTool` muß dann nach jeder Ausführung von Undo oder Redo seine aktuelle Version der `figure` durch die zurückgegebene ersetzen. Wenn für die Undo- oder Redo-Funktion mehrere Materialien rekonstruiert werden müssen, ist die Übergabe als Rückgabeparameter nicht ohne weiteres möglich. Entweder wird für alle veränderten Materialien ein neuer Datentyp definiert und ein Exemplar von diesem zurückgegeben, oder es müßten in der allgemeinen Manipulatorklasse, und somit auch in allen Manipulatoren, Methoden vorhanden sein, mit denen das `toolBasicModellingTool` nach dem Aufruf von Undo oder Redo alle verwendeten Materialien abfragen und auf den aktuellen Stand bringen kann. Aus softwaretechnischer Sicht sind beide Lösungen nicht besonders sauber.
Das Hauptproblem dieser Lösungsvariante ist aber ihr verschwenderischer Umgang mit Systemressourcen. Für jede kleine Änderung müssen zwei vollständige Versionen aller bearbeiteten Materialien gesichert werden. Neben dem hohen Speicherplatzverbrauch für die Sicherung ist hier auch der Rechenaufwand, der für die Erzeugung der Materialkopien anfällt, zu nennen. Beides fällt um so mehr ins Gewicht, je aufwendiger das oder die Materialien sind. Diese Lösung läßt sich optimieren, da die von einer Aktion für ihre Undo-Funktion benötigten Daten mit den Daten übereinstimmen, die von der zuvor ausgeführten Aktion für ihre Redo-Funktion gesichert wurden. Wenn man dies berücksichtigt, reicht es aus, bei jeder ausgeführten Aktion nur eine neue vollständige Sicherung aller Materialien zu erzeugen. Das reduziert zwar den Speicherplatzverbrauch und den benötigten Rechenaufwand um die Hälfte – trotzdem sind diese immer noch wesentlich größer als bei den anderen vorgestellten Lösungsvarianten.

Lösungsvariante 3: Sicherung der Änderungen des Materials im Manipulator

Als ressourcensparende Alternative zur Sicherung des vollständigen Materials bietet es sich an, bei jeder Aktion nur die Bestandteile des Materials zu speichern, die sich verändern. Es kann sich dabei um einzelne Variablen, aber auch um komplexe Objekte handeln. Von zu sichernden Objekten muß, wie in Lösungsvariante 2, eine tiefe Kopie erzeugt werden.
Im Manipulator werden bei der Ausführung der `manipulate()`-Methode von allen Variablen und Objekten, die in dieser Methode verändert werden, zwei Kopien erzeugt. In je einer Kopie wird der Wert einer Variablen bzw. der Zustand eines Objekts vor bzw. nach Ausführung der Methode innerhalb des Manipulators gespeichert. In der `undo()`- bzw. der `redo()`-Methode wird mit Hilfe dieser Variablen und Objekte der Zustand vor bzw. nach dem aktuellen Zustand der Anwendung wiederhergestellt.

Der Vorteil dieser Lösung liegt in dem gegenüber Lösungsvariante 2 wesentlich geringeren Speicherplatzbedarf, den jede für die Undo/Redo-Funktion gesicherte Aktion erfordert. Zudem besteht nicht das Problem, daß sich durch die Undo- oder Redo-Funktion die Identitäten der bearbeiteten Materialien verändern. Auch nach deren Ausführung existiert nur eine Version jedes Materials, da in der `undo()`- bzw. `redo()`-Methode nur einzelne Teile des Materials, nicht aber das Material selbst, ausgetauscht werden.
Ein großer Nachteil dieser Lösungsvariante ist allerdings, daß an den fachlichen Methoden vorbei auf die bearbeiteten Materialien zugegriffen werden muß. Jedes Material muß um Methoden erweitert werden, mit denen die Variablen ausgelesen und gesetzt werden können, die für die Undo/Redo-Funktion kopiert und im Manipulator gesichert werden müssen. Damit wird die fachliche Kapselung des Materials durchbrochen, die normalerweise dafür sorgt, daß es nur mit den dafür vorgesehenen Methoden bearbeitet werden kann. Die für die Undo/Redo-Funktion benötigten Setz- und Auslesemethoden können dazu mißbraucht werden, direkt auf einzelne Teile des Materials zuzugreifen.

4.3.2 Sicherung der Undo/Redo-Informationen in einem eigenen Objekt

Alle bisher vorgestellten Lösungsvarianten basieren auf der Idee, für jede Aktion einen kompletten Manipulator abzuspeichern, in dem die für Undo- und Redo-Funktion notwendigen Daten gesichert sind. Nur der Umfang dieser Daten variiert bei den einzelnen Lösungen. Für kleine, einfach aufgebaute Manipulatoren kann diese Lösung sinnvoll verwendet werden. Wenn die Manipulatoren aber komplizierter und somit umfangreicher sind, müssen bei der Sicherung jedes Manipulator-Objekts zu viele Daten mitgespeichert werden, die für die Rekonstruktion des Materials nach einem Undo- oder Redo-Aufruf nicht benötigt werden. Neben den für Undo und Redo benötigten Kopien des Materials oder einzelner Teile davon, beinhaltet jedes gesicherte Manipulator-Objekt die vollständige `manipulate()`-Methode sowie alle weiteren vorhandenen Methoden einschließlich der von ihnen benutzten Variablen und Objekte. Für die Sicherung jeder Aktion wird daher bei allen bisher vorgestellten Lösungen unnötig viel Speicherplatz belegt.

Lösungsvariante 4: Verwendung einer Klasse pro Manipulator

Um das geschilderte Problem zu umgehen, ist die Idee dieser Lösungsvariante, nur die wirklich für die Undo- und Redo-Funktion notwendigen Daten in einem eigenen Objekt zu sichern. Dieses Objekt hat den Namen „Manipulation“, da in ihm die Auswirkungen eines Manipulators gespeichert werden. In dieses Manipulation-Objekt werden außerdem alle Methoden, die für Undo und Redo relevant sind, verschoben. Für jeden Manipulator muß eine eigene individuelle Manipulation-Klasse implementiert werden, da sich die `undo()`- und die `redo()`-Methode sowie die zu sichernden Variablen und Objekte je nach Manipulator unterscheiden. Die Manipulation-Schnittstelle legt für alle Manipulation-Klassen die zu implementierenden Methoden fest. Sie sind im folgenden aufgeführt:

`public Object undo()`	- führt Undo aus und gibt das aktuelle Material zurück
`public Object redo()`	- führt Redo aus und gibt das aktuelle Material zurück
`public boolean isUndoable()`	- fragt ab, ob die Ausführung von Undo oder Redo erlaubt ist
`public String manipulationDescription()`	- liefert einen Beschreibungstext der zugehörigen Aktion

Für jede ausgeführte Aktion eines Manipulators wird von diesem ein neues Exemplar seiner zugehörigen Manipulation-Klasse erzeugt. Dabei werden die Variablen und Objekte, die in

dieser Manipulation-Klasse für die Undo/Redo-Funktion gesichert werden sollen, als Parameter des Manipulation-Konstruktors übergeben. Wie in den vorigen Lösungsvarianten werden auch hier zwei Versionen von allen bearbeiteten Materialien gesichert. Diese Kopien werden im jeweiligen Manipulator vor bzw. nach der Ausführung der `manipulate()`-Methode erzeugt.
Das neu erzeugte Manipulation-Objekt muß anschließend in der globalen Befehlsgeschichte, die vom übergeordneten Werkzeug (hier dem `toolBasicModellingTool`) verwaltet wird, gespeichert werden. Damit das `toolBasicModellingTool` Zugriff auf das im Manipulator erzeugte Manipulation-Objekt bekommt, wurde die allgemeine Manipulator-Klasse (und damit auch alle von ihr abgeleiteten Manipulator-Klassen) um eine Zugriffsmethode auf das zuletzt erzeugte Manipulation-Objekt erweitert. Diese Methode (`manipulator.manipulation()`) wird jedesmal vom `toolBasicModellingTool` an dem Manipulator aufgerufen, an dem zuvor die `manipulate()`-Methode ausgeführt wurde. Das zurückgelieferte Manipulation-Objekt wird in der Befehlsgeschichte, bei der es sich jetzt statt einer Manipulator-Liste um eine Manipulation-Liste handelt, gespeichert (`_manipulationList.storeManipulation (...)`). Damit kann die Aktion, die in diesem Objekt gespeichert ist, von der Undo/Redo-Funktion zurückgenommen und wiederhergestellt werden.
Die Ausführung von Undo und Redo unterscheidet sich nicht von der zuvor vorgestellten Lösungsvariante. Aus dem Hauptprogramm wird an dem Manipulation-Objekt, das sich vor der Position befindet, an der der Zeiger der Befehlsgeschichte steht, `undo()` aufgerufen und damit die Anwendung in den Zustand vor der Ausführung der letzten Aktion zurückversetzt. Entsprechend wird `redo()` an dem Manipulation-Objekt, auf das der Zeiger deutet, aufgerufen, um die Anwendung in einen späteren Zustand, der zuvor durch die Undo-Funktion verlassen wurde, zurückzuversetzen.

Lösungsvariante 5: Verwendung einer Klasse für alle gleich aufgebauten Manipulatoren

Bei der zuvor beschriebenen Lösungsvariante 4 wurde als Wiederherstellungsgrundlage für alle Manipulatoren jeweils eine Kopie des bearbeiteten Materials vor und nach Ausführung der `manipulate()`-Methode erzeugt. Diese Kopien wurden in einer zum jeweiligen Manipulator gehörenden Manipulation-Klasse zusammen mit den Methoden zur Wiederherstellung gespeichert. Da im Graphical Modelling Tool bei allen Manipulatoren das gleiche (fachlich gesehen sogar dasselbe – allerdings wechselt die Objekt-ID nach Undo und Redo) Material gesichert wird und auch die Implementierungen der `undo()`- und `redo()`-Methoden identisch sind, bietet es sich an, für alle Manipulatoren nur eine einzige Manipulation-Klasse zu verwenden. Diese Klasse, die `FigureManipulation`-Klasse, kann von allen Manipulatoren des Graphical Modelling Tools zur Sicherung ihrer Undo/Redo-Informationen benutzt werden. Die Architektur der für die Undo/Redo-Funktion des Graphical Modelling Tool benötigten Klassen ist in Abbildung 4.5 dargestellt.
Die Klassen `Manipulator`, `ManipulationList` und die Schnittstelle `Manipulation` sind ohne Anpassung allgemein verwendbar. Sie können dem Entwickler als Bestandteil eines Rahmenwerks zur Verfügung gestellt werden. Die Klassen `AddElementManipulator` und `MoveElementManipulator` sind Manipulatoren des Graphical Modelling Tools, die als gemeinsame Klassen zur Speicherung und Wiederherstellung die Klasse `FigureManipulation` verwenden. Diese Klasse muß, wie auch die Manipulatoren, vom Entwickler der Anwendung erstellt werden. `AnotherManipulator` ist eine Beispielklasse für einen Manipulator, der eine eigene Klasse (`AnotherManipulation`) zur Speicherung und Wiederherstellung seiner Daten benötigt.
Wenn in einer Anwendung verschiedene Manipulatoren das gleiche oder die gleichen Materialien bearbeiten und somit gleiche Methoden zu deren Wiederherstellung verwenden, kann

auf die Entwicklung individueller Manipulation-Klassen verzichtet werden. Durch den Einsatz einer gemeinsam verwendeten Manipulation-Klasse kann somit ein Teil des Entwicklungsaufwands eingespart werden.

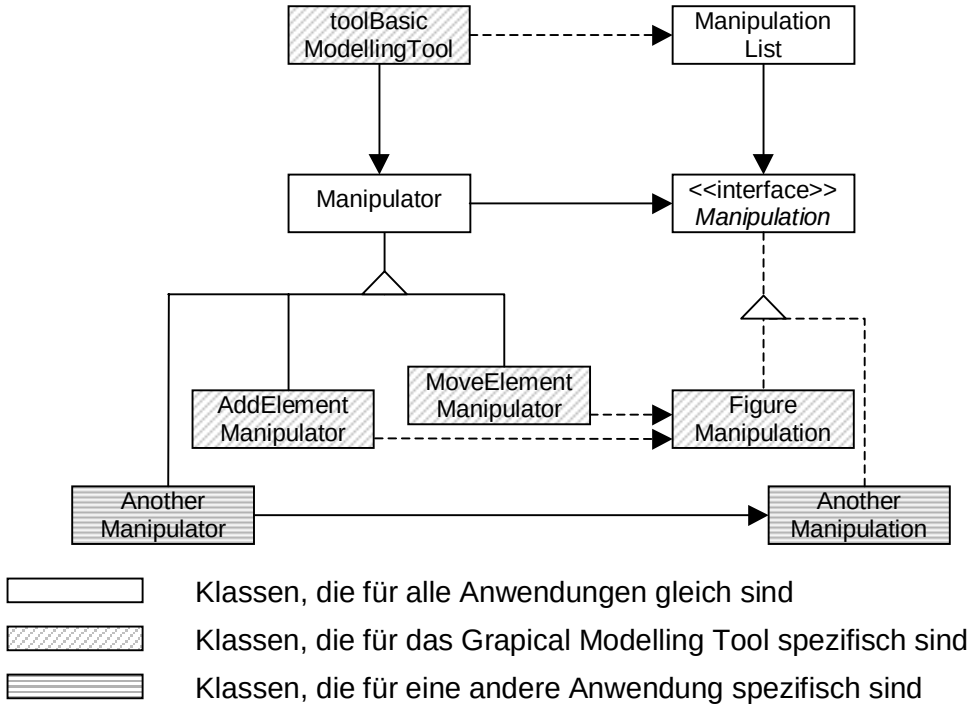

Abbildung 4.5: Aufbau der Undo/Redo-Funktion des Graphical Modelling Tools

4.4 Zusammenfassung

In diesem Kapitel wurden verschiedene Varianten der Undo-Funktion für das Graphical Modelling Tool vorgestellt. Anhand dieser Lösungsvarianten wurden die Vor- und Nachteile der unterschiedlichen Implementierungen herausgearbeitet. Dabei stellte sich heraus, daß die von Gamma vorgeschlagene Lösung zur Realisierung einer Undo-Funktion, bei der jeweils das komplette Befehls- bzw. Manipulatorobjekt gespeichert wird (siehe Kapitel 3.1.2), wenig praktikabel ist. Im Verhältnis zu den für die Rekonstruktion nach Ausführung von Undo oder Redo tatsächlich benötigten Informationen werden bei dieser Lösungsvariante zu viele Daten gesichert. Um dieses Problem zu umgehen, wurde eine neue Klasse eingeführt, die nur für die Rekonstruktion relevante Daten und Methoden beinhaltet.
Außerdem problematisch ist der Zugriff auf die für die Rekonstruktion zu sichernden Daten. Da meist keine fachlichen Methoden vorhanden sind, die zu einer zuvor ausgeführten Aktion das exakte Gegenteil bewirken, besteht nur die Möglichkeit, entweder das komplette Material zu sichern oder dessen fachliche Kapselung aufzubrechen, um mit Hilfe von Zugriffsmethoden direkt einzelne Variablen oder Objekte auszulesen. Aus softwaretechnischer Sicht ist es unsauber, an den fachlichen Methoden vorbei auf das Material zuzugreifen. Es ist allerdings wesentlich aufwendiger in bezug auf Speicherplatzverbrauch und Rechenzeit, bei jeder Aktion ein komplettes Material zu sichern. Bei aufwendigeren Materialien sollte diese Implementierungsvariante daher nicht benutzt werden.
Die Implementierung läßt sich optimieren, indem für gleich aufgebaute Manipulatoren dieselben Klassen zur Speicherung der Undo- und Redo-Informationen verwendet werden. Wenn die Anwendung viele ähnlich oder gleich aufgebaute Manipulatoren verwendet, kann dadurch

die Anzahl der für die Undo-Funktion zu implementierenden Manipulation-Klassen erheblich reduziert werden.
Das wichtigste Kriterium, um eines der vorgestellten Konzepte in großen Anwendungen einsetzen zu können, ist der Umfang der für jeden ausgeführten Arbeitsschritt zu sichernden Daten. Hier haben die ersten drei Varianten den Nachteil, daß immer ein vollständiger Manipulator gesichert werden muß. In den Lösungsvarianten 4 und 5 besteht dieser Nachteil nicht – dafür muß aber eine eigene Manipulation-Klasse implementiert werden. Wenn verschiedene Manipulatoren auf dem gleichen Material arbeiten, kann in Lösungsvariante 5 dieselbe Manipulation-Klasse für die Speicherung verwendet und damit der Entwicklungsaufwand reduziert werden. Von den vorgestellten Lösungen ist daher die Variante 5 die geeignetste für die Integration einer Undo-Funktion in manipulatorbasierte Werkzeuge, da sie einen guten Kompromiß aus Entwicklungsaufwand und Ressourcenverbrauch darstellt.
Allerdings ist der Einsatz dieser, wie auch aller anderen in diesem Kapitel vorgestellten Lösungen, auf eine bestimmte Klasse von Werkzeugen beschränkt. Sie können nur in Werkzeugen eingesetzt werden, die auf Manipulatoren oder dem ähnlich aufgebauten Befehlsmuster basieren.
Anwendungen, die diesem Aufbau nicht entsprechen, müssen erst mit großem Aufwand an die für dieses Undo-Konzept erforderliche Architektur angepaßt werden. Bei bestehenden Anwendungen, die nachträglich um eine Undo-Funktion erweitert werden sollen, ist solch eine Anpassung mit vertretbaren Aufwand oft nicht möglich. Für diese Anwendungsfälle ist das im nächsten Kapitel vorgestellte allgemeine Undo-Konzept besser geeignet. In der folgenden Tabelle werden die Vor- und Nachteile der verschiedenen Konzepte nocheinmal gegenübergestellt:

Lösungsvariante	1	2	3	4	5
Umfang der zu sichernden Daten	o	-	o	+	+
Anwendbarkeit des Konzepts	-	o	+	+	o
einmaliger Entwicklungsaufwand	o	-	o	-	-
wiederkehrender Entwicklungsaufwand	o	o	o	-	+
Probleme mit Materialidentitäten	nein	ja	nein	nein	nein
Umgehung der fachlichen Kapselung	nein	nein	ja	ja	ja

Abbildung 4.6: Vergleich der Lösungsvarianten der Undo-Konzepte für Manipulatoren

5 Undo/Redo-Konzepte für Werkzeuge mit IAK/FK-Trennung

Am Arbeitsbereich Softwaretechnik wurde im Frühjahr 2002 ein Projekt gestartet, in dem mehrere Diplom- und Studienarbeiten unter dem Dach eines gemeinsamen Oberthemas erstellt wurden. Ziel dieses Projekts war es, die in den Diplom- und Studienarbeiten entwickelten Konzepte innerhalb einer komplexen, im praktischen Einsatz befindlichen Anwendung einzusetzen. Bisher wurden viele der erarbeiteten Ideen nur anhand von kleinen Beispielprogrammen umgesetzt, so daß sich kaum Aussagen über die praktische Einsetzbarkeit machen ließen. Als gemeinsames Oberthema wurde der Integrationsserver gewählt. Es handelt sich hierbei um eine Anwendung, mit der mehrere Entwickler gemeinsam an einem Softwareprojekt arbeiten und gleichzeitig die Integrität dieses Projekts aufrecht erhalten können (siehe Abbildung 5.1). Mit Hilfe des Integrationsservers wird jede Änderung am Quellcode allen anderen Entwicklern zugänglich gemacht und gleichzeitig sichergestellt, daß nur erfolgreich getesteter Programmcode in das Projekt gelangt. Die erste Version des Integrationsservers wurde von Robert Tunkel im Rahmen seiner Studienarbeit entwickelt (siehe [Tun01]).

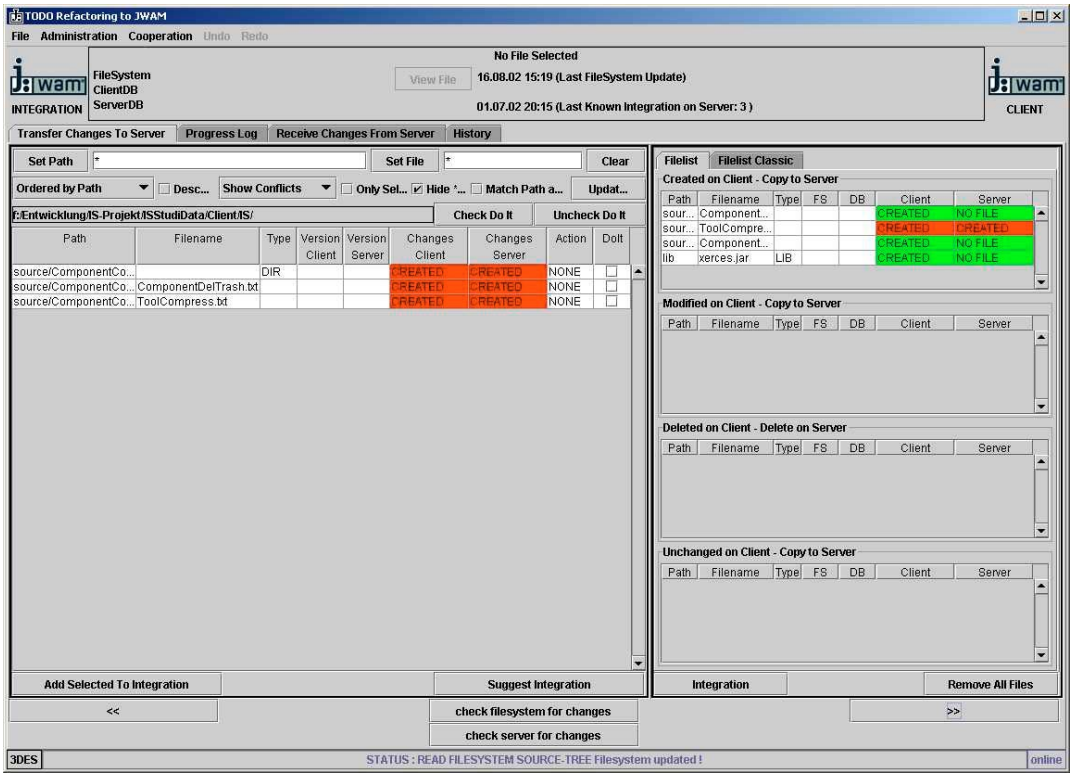

Abbildung 5.1: Der Integrationsserver in der im Diplomarbeiterprojekt entwickelten Version

Im Verlauf des Studien- und Diplomarbeiterprojekts sollte der Integrationsserver mit dem Schwerpunkt auf den Themen der einzelnen Arbeiten umgebaut und erweitert werden. Eine dieser Erweiterungen lag darin, den Integrationsserver um eine lokale Undo- und Redo-Funktion zu erweitern. Dabei sollte es beispielsweise möglich sein, Änderungen an der Zusammenstellung der zu integrierenden Dateien oder an dem Beschreibungstext für eine Integration rückgängig zu machen.
Die Grundlage der zu entwickelnden Undo-Funktion sollte das in Kapitel 4 vorgestellte Konzept bilden. Allerdings stellte sich heraus, daß es die bestehende Architektur des Integrationsservers nicht ohne weiteres erlaubte, dieses Undo-Konzept zu integrieren. Die Architekturen des Integrationsservers und des Graphical Modelling Tools, für das dieses Undo-Konzept

ursprünglich entwickelt wurde, unterscheiden sich zu sehr, um es direkt übertragen zu können.
Das Graphical Modelling Tool ist ein Werkzeug, das aus einer zentralen Kernklasse sowie einer größeren Anzahl von Manipulatorklassen besteht. In den Manipulatorklassen werden jeweils vom Benutzer ausführbare Aktionen implementiert. Bei diesen Aktionen handelt es sich um isolierte Operationen auf einer relativ kleinen Anzahl von Objekten, die zudem frei von Seiteneffekten sind. Damit kann für alle Aktionen die Undo-Funktion innerhalb ihrer Manipulatorklasse implementiert werden, ohne daß dabei die Gefahr besteht, daß Teile der Anwendung nicht in ihren früheren Zustand zurückversetzt werden, wenn Undo ausgeführt wird.
Im Gegensatz dazu besteht der Integrationsserver aus einer großen Anzahl von Werkzeugen, die zum Teil eigenständig und zum Teil als Subwerkzeuge anderer Werkzeuge implementiert sind. Diese Werkzeuge besitzen, im Gegensatz zu den Manipulatoren des Graphical Modelling Tools, eine relativ große Funktionalität. Daher müssen in einigen Werkzeugen mehrere Undo-Funktionen implementiert werden, die jeweils auf eine Aktion angepaßt sind. Allerdings müssen im Integrationsserver nicht für jede Aktion Undo-Funktionen entwickelt werden. Viele der vom Integrationsserver für den Anwender bereitgestellten Aktionen ändern nur die Sicht auf ein Material und müssen somit von der Undo-Funktion nicht unterstützt werden.
Ein weiteres Problem bei der Entwicklung der Undo-Funktion liegt darin, daß viele der einzelnen Werkzeuge, wie es der Werkzeug- und Materialansatz vorsieht, in Interaktions- und Funktionskomponenten getrennt sind. Bei diesen Werkzeugen muß besonders darauf geachtet werden, daß auch die Darstellung aktualisiert wird, wenn der Benutzer Undo oder Redo ausführt und damit eine fachliche Aktion zurücknimmt.
Um das zuvor vorgestellte Konzept verwenden zu können, müßte die Architektur des Integrationsservers so angepaßt werden, daß sie der des Graphical Modelling Tools entspricht. Dafür wäre es notwendig, alle Aktionen, die von der Undo-Funktion abgedeckt werden sollen, in eigene Klassen auszulagern. Diese komplette Umstellung auf eine befehls- oder manipulatorbasierte Architektur wäre aufgrund der Programmgröße (ca. 200 Klassen zu Projektbeginn) nur mit sehr großem Zeitaufwand möglich gewesen. Da parallel aber an anderen Teilen des Integrationsservers weiterentwickelt werden mußte und es zudem die Vorgabe gab, nur vollständig funktionsfähigen Programmcode in das Projekt zu integrieren, mußte eine Lösung verwendet werden, die eine schrittweise Einführung der Undo-Funktion erlaubte und zugleich die bestehenden Schnittellen der einzelnen Klassen möglichst unverändert ließ.
In diesem Kapitel wird ein Undo-Konzept vorgestellt, das die genannten Vorgaben erfüllt. Es kann unabhängig von der eingesetzten Werkzeugkonstruktion verwendet werden, erfordert dafür aber mehr Anpassungsaufwand vom Entwickler. Zunächst wird die Grundidee vorgestellt, auf der das Undo-Konzept beruht. Anschließend wird auf die `UndoData`-Liste, die der zentrale Behälter zur Speicherung aller Wiederherstellungsinformation ist, eingegangen. Im letzten Teil dieses Kapitels werden einige funktionelle Erweiterungen beschrieben, um die das Undo-Konzept im Laufe seiner Entwicklung ergänzt wurde.

5.1 Grundidee des Undo/Redo-Konzepts für IAK/FK-Werkzeuge

Für die Entwicklung einer Undo-Funktion kam aus den bereits genannten Gründen eine Umstellung des Integrationsservers auf eine befehls- oder manipulatorbasierte Architektur nicht in Frage. Daher mußte für die Realisierung der Undo-Funktion ein Konzept verwendet werden, das mit der bestehenden Architektur des Integrationsservers zusammenarbeiten kann.
Die vom Integrationsserver bereitgestellten Aktionen sind als Aufruf einzelner Methoden an den Funktionskomponenten der verschiedenen (Sub-)Werkzeuge realisiert. Die Einordnung

dieser Methoden innerhalb der Funktionskomponenten sollte, egal ob für sie eine Undo-Unterstützung entwickelt wird oder nicht, unverändert bleiben. Zudem sollten die Signaturen dieser Methoden nicht verändert werden, da sonst Änderungen an allen Stellen, an denen sie aufgerufen werden, nötig sind. Das unter diesen Vorgaben entwickelte Konzept zur Realisierung der Undo/Redo-Funktion für den Integrationsserver ist wie folgt aufgebaut:
Um eine Aktion des Integrationsservers später rückgängig machen oder erneut ausführen zu können, werden, ähnlich wie in den Lösungen aus Kapitel 4, die Zustände der von dieser Aktion bearbeiteten Materialien einmal vor und einmal nach ihrer Ausführung gesichert (vgl. Abbildung 4.3). Wenn die Ausführung der Methode unabhängig vom Ausgangszustand zum gleichen Ergebnis führt, wird nur die erste Sicherung für die Undo-Funktion benötigt. Diese Sicherungen können entweder nur aus den veränderten Daten oder aus den vollständigen Materialien bestehen.
Die Undo/Redo-Informationen werden in speziell für diesen Zweck entwickelten Klassen gesichert, die alle eine gemeinsame Schnittstelle implementieren (`UndoData`, siehe Abbildung 5.2). In dieser Schnittstelle sind die Methoden vorgegeben, mit denen später die gesicherten Versionen der Materialien rekonstruiert werden können. Für jede ausgeführte Aktion, die Undo und Redo unterstützen soll, wird ein neues Exemplar einer für diese Aktion entwickelten `UndoData`-Klasse erzeugt und in ihr die Undo/Redo-Informationen gesichert. Anschließend werden diese `UndoData`-Objekte in einer gemeinsamen Liste, der Befehlsgeschichte, gespeichert.

```
public interface UndoData
{
   public void undo();
   public void redo();
   public String undoDescription();
   public void setUndoDescription(String undoDescription);
   public boolean isUndoAndRedoable();
   public void setUndoAndRedoable(boolean isUndoAndRedoable);
   public boolean isSeparateUndoStep();
   public void setSeparateUndoStep(boolean isSeparateUndoStep);
   public boolean canBeAccumulated();
   public void setCanBeAccumulated(boolean canBeAccumulated);
   public boolean isAccumulationCompleted();
   public void setAccumulationCompleted
               (boolean accumulationCompleted);
}
```

Abbildung 5.2: Schnittstelle UndoData

Um die letzte ausgeführte Aktion rückgängig zu machen – also den Zustand vor der Ausführung dieser Aktion wiederherzustellen – muß an dem zugehörigen `UndoData`-Objekt die dafür durch die `UndoData`-Schnittstelle vorgegebene Rekonstruktionsmethode (`undo()`) aufgerufen werden. Entsprechend muß, wenn die zuletzt zurückgenommene Aktion wiederhergestellt werden soll, die `redo()`-Methode aufgerufen werden. Das `UndoData`-Objekt, an dem diese beiden Methoden aufgerufen werden, muß jeweils zuvor an der Befehlsgeschichte erfragt werden.
Wie bereits erwähnt wurde, müssen für Methoden, die mit Hilfe der Undo/Redo-Funktion zurückgenommen oder wiederhergestellt werden sollen, jeweils auf sie angepaßte Versionen der `UndoData`-Klasse entwickelt werden. In diesen Klassen werden für alle Variablen und Objekte, die für die Undo- und die Redo-Funktion der zugehörigen Aktion gesichert werden müssen, ihnen entsprechende Variablen bzw. Objekte definiert. Bei der Erzeugung von Exemplaren dieser Klassen werden die zu sichernden Variablen bzw. Objekte als Parameter ihrer

Konstruktoren übergeben und den in der Klasse definierten Variablen bzw. Objekten zugewiesen.
In der `undo()`-Methode muß die Funktionalität implementiert werden, mit der die zu dieser Klasse gehörende Aktion rückgängig gemacht werden kann. Entsprechend muß in der `redo()`-Methode implementiert werden, wie diese Aktion wiederhergestellt werden kann, wenn sie zuvor rückgängig gemacht wurde. Die `undoDescription()-Methode` liefert eine textuelle Beschreibung dieser Aktion, die z.B. als Hilfestellung für den Benutzer dienen kann.

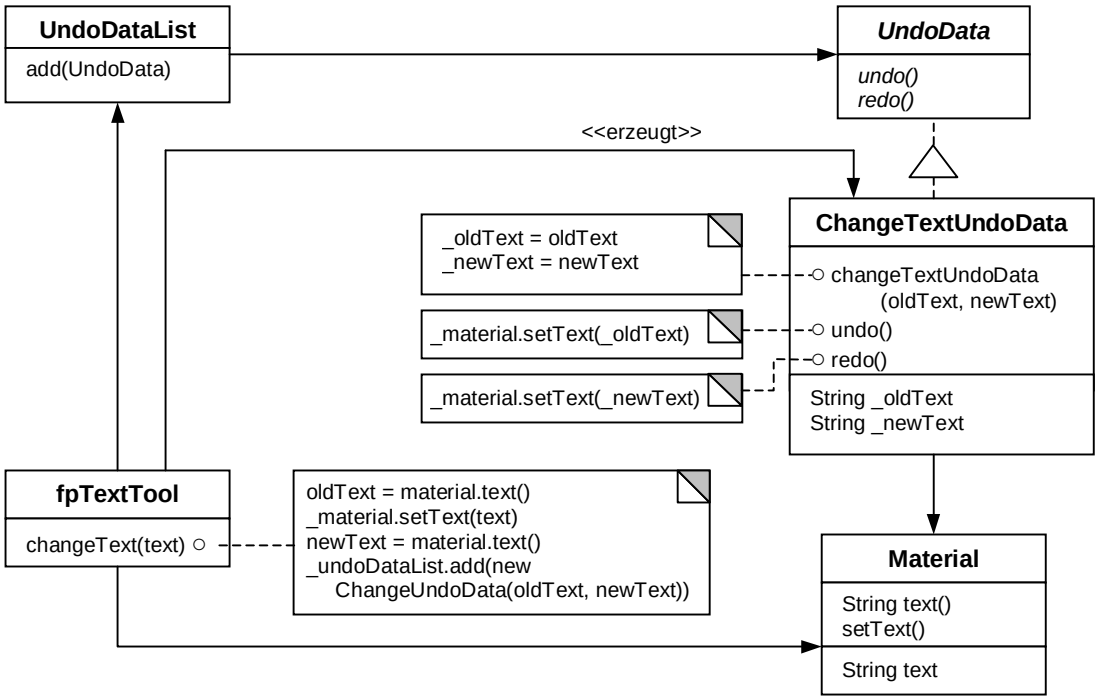

Abbildung 5.3: Klassendiagramm eines Beispiels zur Benutzung von UndoData-Klassen

Mit `setUndoDescription(...)` kann diese Beschreibung zugewiesen werden. Die übrigen Methoden werden für Erweiterungen des Grundkonzepts benötigt, die in Kapitel 5.3 vorgestellt werden.
Eine `UndoData`-Klasse kann auch für die Sicherung der Zustände verschiedener Aktionen verwendet werden, wenn bei allen Aktionen Objekte oder Variablen mit den gleichen Typen zu sichern sind (vgl. Kapitel 4.3.2 Lösungsvariante 5). Dann bietet es sich an, als zusätzlichen Parameter die Beschreibung (`UndoDescription`) der Aktion mit in den Konstruktor der `UndoData`-Klasse aufzunehmen und bei der Erzeugung eines neuen Objekts diesen Parameter zuzuweisen. Bei `UndoData`-Klassen, die nur zur Sicherung einer Aktion verwendet werden, kann die Beschreibung auch fest in dieser Klasse vorgegeben werden.
Für die Schnittstelle `UndoData` existiert eine vorgegebene Standardimplementierung (`UndoDataImpl`), die als Oberklasse für alle konkreten `UndoData`-Klassen dient. In dieser Klasse sind die Methoden implementiert, die für alle `UndoData`-Klassen identisch sind und daher nicht jedesmal neu entwickelt werden müssen.
Um die Anzahl eigenständiger Klassen und somit die Anzahl der Dateien innerhalb eines Projekts nicht unnötig zu vergrößern, werden alle `UndoData`-Klassen als innere Klassen implementiert. Da die meisten `UndoData`-Klassen einen relativ kleinen Codeumfang haben, bietet es sich an, diese Klassen innerhalb der Funktionskomponente des Werkzeugs zu dem sie gehören, zu plazieren. Damit wird zum einen verhindert, daß für jede Aktion, die Undo und Redo unterstützt, eine eigene Datei notwendig ist und zum anderen wird die Übersichtlichkeit gesteigert, da die `UndoData`-Klassen genau an der Stelle im Programmcode stehen,

an der sie erzeugt und benutzt werden. Zur Vereinheitlichung der Benennung setzen sich die Namen aller `UndoData`-Klassen aus der Methode, zu der sie gehören, und der Endung „UndoData“ zusammen.
In Abbildung 5.3 ist das Klassendiagramm eines einfachen Beispielwerkzeugs dargestellt, aus dem die grundlegende Funktionsweise der in diesem Kapitel vorgestellten Undo-Lösung ersichtlich ist. Es sind nur die für die Undo-Funktion relevanten Klassen und Methoden eingezeichnet. Das Werkzeug besitzt eine Methode, mit der in einem Material ein Text geändert werden kann (`changeText(text)`). Mit jedem Aufruf dieser Methode werden eine Sicherung des alten und eine des veränderten Textes erzeugt. Diese beiden Sicherungen werden in einem neu erzeugten `UndData`-Objekt in einer `UndoData`-Liste gespeichert. Mit Hilfe der Undo und der Redo Methode kann zu einem späteren Zeitpunkt der Text vor bzw. nach der Änderung wiederhergestellt werden.
Zur Speicherung aller `UndoData`-Objekte wird in jedem Werkzeug eine einzige zentrale `UndoData`-Liste (siehe Kapitel 5.2) verwendet. Auf diese Liste kann bei größeren Werkzeugen, die aus verschiedenen Subwerkzeugen aufgebaut sind, nicht direkt zugegriffen werden. Der Grund dafür ist, daß Subwerkzeuge im allgemeinen keine Informationen darüber besitzen, in welches Werkzeug sie eingebettet sind.
Eine mögliche Lösung für dieses Problem wäre es, alle Funktionskomponenten der Subwerkzeuge um Ereignisse (Events) zu ergänzen, die ausgelöst werden, wenn ein zu speicherndes `UndoData`-Objekt vorliegt. An diesen Ereignissen kann sich das übergeordnete Werkzeug anmelden und, nachdem es durch das Subwerkzeug ausgelöst wurde, dieses Objekt abfragen. Bei größeren Werkzeugen, wie z.B. dem Integrationsserver, ist diese Lösung allerdings weniger geeignet. Da sich das Hauptwerkzeug bei allen Subwerkzeugen, in die wiederum weitere Subwerkzeuge eingebettet sein können, anmelden muß, würde die entstehende hierarchische Ereignisstruktur den Programmablauf sehr unübersichtlich und zudem unnötig kompliziert machen.
Als alternative Lösungsmöglichkeit bietet es sich an, einen Request zu verwenden. Ein Request ist eine an das Entwurfsmuster der Zuständigkeitskette (Chain of Responsibility) angelehnte Konstruktion, mit der Empfänger und Auslöser einer Anfrage entkoppelt werden können (vgl. [Gam96]). Dabei wird im Subwerkzeug ein spezielles Request-Objekt erzeugt, bei Bedarf parametrisiert und einem Request-Handler zur Bearbeitung übergeben. Der Request-Handler ist ein im Gesamtwerkzeug nur einmal vorhandenes Objekt, das alle Requests entgegen nimmt und abhängig von ihren Typen und ihren Parametern verschiedene Aktionen veranlaßt.
Für die Undo/Redo-Unterstützung im Integrationsserver existiert ein Request (`reqStoreUndoData`), dem das zu speichernde `UndoData`-Objekt als Parameter des Konstruktors übergeben wird. Dieser Request wird mit der `storeUndoData()`-Methode erzeugt und anschließend abgeschickt. Die Bearbeitung aller Requests übernimmt im Integrationsserver die Werkzeugklasse des Hauptwerkzeugs (`toolIntegrationClient`). Diese sorgt dafür, daß das im Request gespeicherte `UndoData`-Objekt in der `UndoData`-Liste gesichert und daraufhin die Benutzungsoberfläche aktualisiert wird.
In dem in Abbildung 5.4 dargestellten Klassendiagramm sind die Beziehungen zwischen den wichtigsten im Integrationsserver für die Undo/Redo-Funktion benutzten Klassen dargestellt. Die mit dem Stereotyp „request“ versehene Beziehung zwischen der `UndoData`-Liste und der Funktionskomponente des Subwerkzeugs deutet an, daß zwischen diesen Klassen keine direkte Benutzungsbeziehung besteht. Informationen werden nur mit Hilfe des Request vom Subwerkzeug über den Umweg der Hauptwerkzeugklasse (in der Abbildung aus Vereinfachungsgründen weggelassen) an die `UndoData`-Liste weitergereicht.

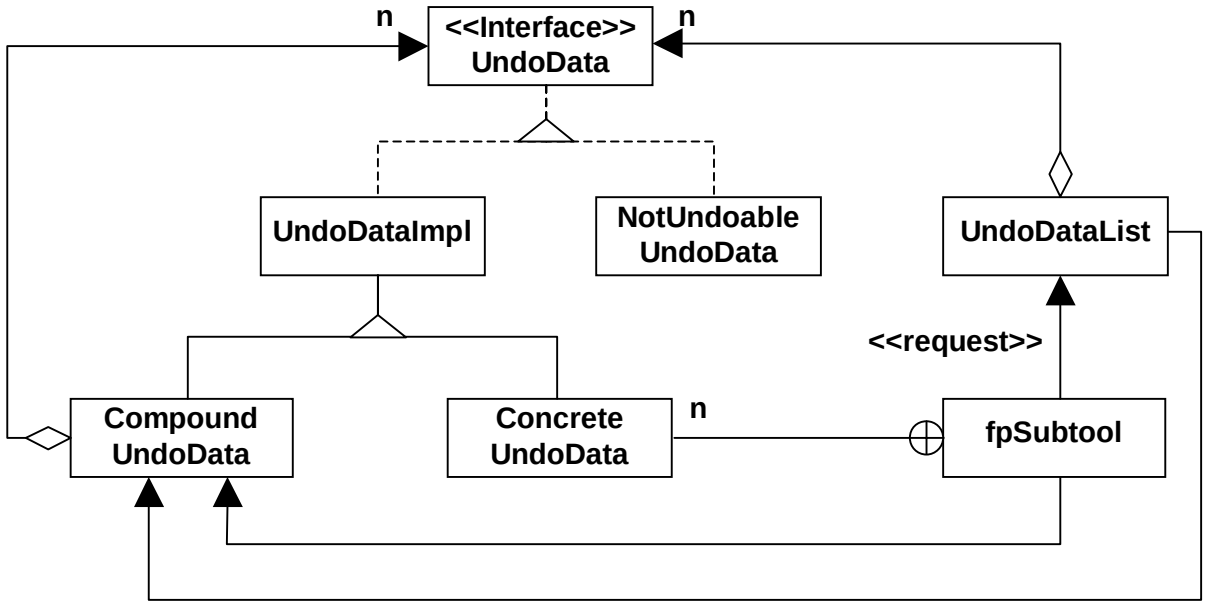

Abbildung 5.4: Aufbau des Undo-Konzepts für den Integrationsserver

5.2 UndoData-Liste

Die zentrale Einrichtung zur Speicherung aller erzeugten `UndoData`-Objekte ist die `UndoData`-Liste. Diese Liste muß in einem Werkzeug, welches Undo unterstützen soll, genau einmal vorhanden sein. Sie sollte möglichst schon beim Start des Hauptwerkzeugs erzeugt werden.
Intern besteht die `UndoData`-Liste aus einer verketteten Liste (`LinkedList`) mit einer konstanten maximalen Größe. Zudem existiert ein Zeiger, der auf die aktuelle Position innerhalb dieser internen Liste zeigt (siehe Abbildung 5.5). Die öffentliche Schnittstelle der `UndoData`-Liste sieht wie folgt aus:

`public` `UndoDataList()`: Konstruktor - erzeugt eine neue leere `UndoData`-Liste und setzt den internen Zeiger an den Anfang dieser Liste

`public void` `add(UndoData)`: fügt ein `UndoData`-Objekt in die interne Liste ein
- hängt das Objekt an das Ende der Liste, wenn der Zeiger hinter dem letzten Objekt steht
- ersetzt das Objekt, auf das der Zeiger deutet und löscht alle darauffolgenden Objekte, wenn der Zeiger nicht am Ende steht
- entfernt das erste Objekt, wenn die Maximalgröße der Liste überschritten wurde
- behandelt in der erweiterten Version Spezialfälle (siehe Kapitel 5.3)

`public` `UndoData getLast()`: liefert das Objekt vor der aktuellen Zeigerposition

`public` `UndoData getNext()`: liefert das Objekt, auf das der Zeiger deutet

`public boolean` `canUndo()`: testet, ob Undo möglich ist (Zeiger nicht am Beginn der Liste)

`public boolean` `canRedo()`: testet, ob Redo möglich ist (Zeiger nicht am Ende der Liste)

`public void` `undoLast()`: ruft die `undo()`-Methode am Objekt vor der aktuellen Zeigerposition auf

`public void` `redoNext()`: ruft die `redo()`-Methode an dem Objekt, auf das der Zeiger deutet, auf

public `String[] getHistoryForUndo()`: liefert eine Liste aller Beschreibungen, für die Undo möglich ist

public `String[] getHistoryForRedo()`: liefert eine Liste aller Beschreibungen, für die Redo möglich ist

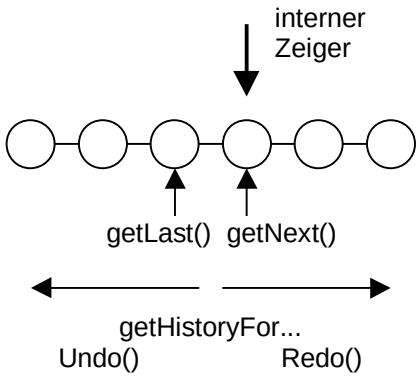

Abbildung 5.5: UndoDataList mit konstanter Größe n = 6

Auch wenn die Liste eine konstante Größe hat, ist eine Implementierung als verkettete Liste (`LinkedList`) gegenüber einer Implementierung als Feldliste (`ArrayList`) vorzuziehen. Da nach Erreichen der Maximalgröße alle Einträge mit jedem Element, das am Ende der Liste hinzugefügt wird, um eine Position nach vorne gelangen, müßten bei einer Feldliste jedesmal alle Einträge umgeschichtet werden. Bei einer verketteten Liste besteht dieser Nachteil nicht.

5.3 Zusätzliche Konzepte

Mit dem bis jetzt vorgestellten Undo-Konzept lassen sich für die meisten einfachen Anwendungen Undo-Funktionen realisieren. Für komplexere Programme sind, wie sich bei der Entwicklung der Undo-Funktion für den Integrationsserver zeigte, allerdings noch einige Erweiterungen dieses Konzepts notwendig.
Die in diesem Unterkapitel vorgestellten zusätzlichen Konzepte entstanden zum Teil direkt aus neuen Anforderungen, die während der Implementierung der Undo-Funktion für den Integrationsserver ersichtlich wurden und mit dem bisherigen Konzept nicht abgedeckt werden konnten. Es wurde die Möglichkeit geschaffen, bestimmte Aktionen explizit als nicht zurücknehmbar zu kennzeichnen und so die Ausführung von Undo und Redo auf diesen Aktionen zu verhindern. Außerdem ist es nun möglich, mehrere einzelne Aktionen zu einer Gesamtaktion zusammenzufassen. Dies kann entweder vom Entwickler veranlaßt werden oder automatisch geschehen. Schließlich kann jetzt zwischen Aktionen, auf die der Benutzer direkt mit der Undo/Redo-Funktion Zugriff hat und Aktionen, die nur infolge weiterer Aktionen zurückgenommen oder wiederholt werden können, unterschieden werden.

5.3.1 Nichtrückgängigmachbare Aktionen

In interaktiven Anwendungen gibt es Aktionen, die von einer Undo-Funktion nicht abgedeckt werden – entweder weil es nicht möglich ist, für sie eine Undo-Funktion zu erstellen oder weil es vom Entwickler nicht gewollt ist. Unter die Aktionen, für die keine Undo-Funktion erstellt werden kann, fallen alle diejenigen, deren Auswirkungen über die Grenzen der benutzten Anwendung hinausgehen. Wenn der weitere Programmablauf von diesen Aktionen abhängt, dürfen die zuvor ausgeführten Aktionen nicht mehr zurückgenommen werden, da sonst die Anwendung in einen inkonsistenten Zustand gelangen kann. Solche Fälle treten zum

Beispiel auf, wenn auf ein nichtlokales Medium, wie etwa eine externe Datenbank, zugegriffen wird, wie in dem folgenden Beispiel: Ein Sachbearbeiter liest aus einer Kundendatenbank, die auf einem Server im Netzwerk liegt, einen Datensatz ein, verändert die Kundenanschrift und speichert die veränderten Daten anschließend wieder in der Datenbank. Die Änderung der Adresse ist eine lokale Aktion, die mit Hilfe der Undo-Funktion zurückgenommen werden kann. Da dies aber keine Auswirkungen auf die in der Datenbank gespeicherte Version des Datensatzes hat, sollte diese Aktion nicht mehr zurückgenommen werden können, nachdem der Datensatz abgespeichert worden ist.
Neben diesen Aktionen gibt es Aktionen, die es explizit vorsehen, daß sie unumkehrbar sind. Nachdem solche Aktionen ausgeführt wurden, dürfen weder diese, noch die Aktionen davor rückgängig gemacht werden.
Aus den genannten Gründen ist es notwendig, das vorgestellte Undo-Konzept um einen Mechanismus zu erweitern, mit dem signalisiert werden kann, daß nach der Ausführung einer Aktion diese und alle vorigen Aktionen nicht mehr zurückgenommen werden dürfen. Technisch kann die weitere Benutzung der Undo-Funktion auf früher ausgeführten Aktionen am einfachsten verhindert werden, indem alle Einträge der Befehlsgeschichte gelöscht werden. Normalerweise würde es reichen, die Befehlsgeschichte um eine neue Methode zu erweitern, mit der dies möglich ist. Da aber im Integrationsserver sämtliche Subwerkzeuge keinen direkten Zugriff auf die Befehlsgeschichte haben, scheidet dieser Lösungsweg aus.
Daher wird stattdessen die Information, daß die Befehlsgeschichte gelöscht werden soll, in den zu speichernden `UndoData`-Objekten mit Hilfe des Request-Mechanismus an die Befehlsgeschichte übergeben. Dazu wird eine Statusvariable benutzt, die von der Befehlsgeschichte abgefragt wird, bevor das übergebene `UndoData`-Objekt gespeichert wird. Wenn die Statusvariable gesetzt ist, wird die Befehlsgeschichte geleert und das übergebene Objekt nicht abgespeichert.
Praktisch wurde dieses Konzept realisiert, indem die Schnittstelle `UndoData`, und damit alle sie implementierenden UndoData-Klassen, um zwei Methoden erweitert wurde, mit denen der Wert der oben genannten Statusvariable (**`boolean`** `_isUndoAndRedoable`) ausgelesen und gesetzt werden kann:

```
public boolean isUndoAndRedoable()
public void setUndoAndRedoable(boolean isUndoAndRedoable)
```

In der Standardimplementierung (`UndoDataImpl`) wird die Statusvariable mit dem Wert `true` initialisiert. Das bedeutet, daß bei allen abgeleiteten Klassen und den von diesen erzeugten Exemplaren der Wert der Statusvariable nur verändert werden muß, wenn die zu diesen Objekten gehörenden Aktionen nicht mit Hilfe der Undo-Funktion zurückgenommen werden sollen. Die beiden Methoden sind in `UndoDataImpl` implementiert und können von allen abgeleiteten Klassen unverändert benutzt werden.
Soll nach der Ausführung einer bestimmten Aktion die Befehlsgeschichte gelöscht werden, muß das von dieser Aktion erzeugte `UndoData`-Objekt zunächst als „nichtrückgängigmachbar“ markiert werden (`UndoData.setUndoAndRedoable(false)`). Anschließend wird dieses Objekt, wie ein normales `UndoData`-Objekt auch, mit Hilfe eines Requests an die `UndoData`-Liste geschickt. Die Liste liest den internen Zustand des Objekts aus und löscht, statt es zu speichern, seine interne Liste und somit die Befehlsgeschichte.
Um unabhängig von bestehenden Klassen die `UndoData`-Liste löschen zu können, gibt es die `NotUndoableUndoData`-Klasse. Objekte dieser Klasse dienen nur dazu, die Befehlsgeschichte zurückzusetzen, haben sonst aber keine weitere Funktionalität. Dazu liefert die `isUndoAndRedoable()`-Methode dieser Klasse immer `false`, alle anderen durch die `UndoData`-Schnittstelle vorgegebenen Methoden sind leer implementiert.

5.3.2 Zusammenhängende Aktionen

Es gibt Aktionen, die aus Benutzersicht nur ein einzelner Arbeitsschritt sind, intern aber aus mehreren Aktionen bestehen und somit mehrere `UndoData`-Objekte zur Speicherung ihres Zustands verwenden. Auch diese Aktionen soll der Benutzer mit nur einmaliger Benutzung der Undo- oder Redo-Funktion zurücknehmen oder wiederholen können.
Um dies zu ermöglichen, wird eine Behälterklasse (`CompoundUndoData`) eingeführt, zu der die `UndoData`-Objekte hinzugefügt werden können, die zusammengefaßt werden sollen. Gleichzeitig implementiert diese Behälterklasse selbst die für alle `UndoData`-Klassen verbindliche Schnittstelle. Daher können ihre Exemplare wie normale `UndoData`-Objekte in der Befehlsgeschichte gespeichert werden. Ein Aufruf von Undo oder Redo an einem in der Befehlsgeschichte gespeicherten Behälterobjekt führt dazu, daß an allen in ihm gespeicherten `UndoData`-Objekten die entsprechenden Methoden ausgeführt werden. Damit werden alle in diesem Objekt gespeicherten Einzelaktionen rückgängig gemacht oder wiederholt.
Die Klasse `CompoundUndoData` ist von der Standardimplementierung aller `UndoData`-Klassen (`UndoDataImpl`) abgeleitet und erfüllt somit die `UndoData`-Schnittstelle. Als einzige zusätzliche Methode besitzt die `CompoundUndoData`-Klasse eine Methode, mit der `UndoData`-Objekte zu ihrer internen Liste, die als einfacher `Vector` realisiert ist, hinzugefügt werden können (**`public`** `void add(UndoData)`).
Bei der Funktionsweise der `undo()`-Methode von `CompoundUndoData` ist zu beachten, daß die `undo()`-Methoden der einzelnen gespeicherten `UndoData`-Objekte in umgekehrter Reihenfolge zu ihrer Speicherung aufgerufen werden. Dies ist notwendig, damit auch die Einzelaktionen in exakt umgekehrter Reihenfolge zu ihrer Ausführung zurückgenommen werden und es somit zu keinen Inkonsistenzen kommt.
Mit Hilfe von `CompoundUndoData`-Objekten lassen sich vom Entwickler aus kleineren Aktionen kompliziertere Aktionen aufbauen, die Undo und Redo unterstützen. Dazu wird ein neues `CompoundUndoData`-Objekt erzeugt und anschließend zu diesem alle zur Gesamtaktion gehörenden `UndoData`-Objekte hinzugefügt. Nachdem das `CompoundUndoData`-Objekt in der `UndoData`-Liste gespeichert worden ist, können alle Einzelaktionen immer als ein einziger Schritt wiederholt oder rückgängig gemacht werden.

5.3.3 Akkumulierbare Aktionen

Das im vorigen Abschnitt vorgestellte Konzept hat den Nachteil, daß mehrere Aktionen nur manuell zusammengefaßt werden können. Der Entwickler muß im Programmcode jedes `UndoData`-Objekt angeben, das zum Behälterobjekt hinzugefügt werden soll. Außerdem besteht das Problem, daß zu Behälterobjekten keine weiteren `UndoData`-Objekte hinzugefügt werden können, nachdem sie abgespeichert wurden. Die Speicherung ist aber die Vorraussetzung dafür, daß auf ihnen Undo oder Redo ausgeführt werden kann. Um auch in diesen Fällen Undo und Redo zu unterstützen, wurde das folgende Konzept entwickelt, mit dem die `UndoData`-Objekte mehrerer gleicher Aktionen automatisch zusammengefaßt werden können. Die Objekte werden Stück für Stück zu einem gemeinsamen Behälterobjekt hinzugefügt – daher der Name „Akkumulationskonzept“.
Für jede Aktionsart, und damit die zugehörige `UndoData`-Klasse, muß zunächst einmal generell festgelegt werden, ob von ihr mehrere nacheinander ausgeführte individuelle Aktionen zu einer einzigen Gesamtaktion zusammengefaßt werden sollen. Zusätzlich kann für jede individuelle Aktion festgelegt werden, ob sie die letzte in einer Reihe zusammenfaßbarer Aktionen ist und somit nach ihrer Ausführung die Gesamtaktion abgeschlossen wird. Für die Kennzeichnung dieser beiden Eigenschaften werden, wie im Konzept für Aktionen, die nicht zu-

rückgenommen werden sollen (vgl. Kapitel 5.3.1), zwei Statusvariablen eingeführt, die in jedem `UndoData`-Objekt vorhanden sein müssen. Wie bereits erwähnt, wird für jede ausgeführte Aktion, die Undo/Redo unterstützt, ein `UndoData`-Objekt an die Befehlsgeschichte übergeben. Hier wird anhand der Statusvariablen und der Beschreibung der Aktion entschieden, wie das zu speichernde `UndoData`-Objekt behandelt wird:
Wenn das Objekt nicht akkumulierbar ist, wird es direkt in der Befehlsgeschichte gespeichert. Anderenfalls wird es zu einem neuen Behälterobjekt hinzugefügt und dieses wird in der Befehlsgeschichte gesichert. Ist das nächste zu speichernde Objekt ebenfalls akkumulierbar und stimmt zudem seine Beschreibung mit der des zuvor gespeicherten Objekts überein (d.h. beide Objekte gehören zur gleichen Aktion), wird das zweite UndoData-Objekt zum selben Behälterobjekt hinzugefügt. Für beide Aktionen existiert dann nur ein Eintrag in der Befehlsgeschichte. Undo und Redo führen dazu, daß, wie im vorigen Abschnitt beschrieben, an allen abgespeicherten Objekten die entsprechenden Methoden aufgerufen werden und alle dazugehörigen Aktionen zusammen zurückgenommen oder wiederholt werden.
Folgen zwei `UndoData`-Objekte mit unterschiedlicher Beschreibung nacheinander, wird für das zweite ein neues Behälterobjekt erzeugt, in dem es in der Befehlsgeschichte gespeichert wird. Das Akkumulationskonzept ist auf gleiche Aktionen beschränkt, da sonst jede Einzelaktion, die prinzipiell akkumuliert werden kann, explizit abgeschlossen werden müßte. Gleichzeitig macht es im allgemeinen keinen Sinn, unterschiedliche Aktionen zusammenzufassen.
Die Möglichkeit, eine Akkumulation manuell abzuschließen, wurde eingeführt, um es dem Entwickler auch bei aufeinanderfolgenden gleichen Aktionen zu ermöglichen, diese Aktionen in mehreren Behälterobjekten zu gruppieren. Wenn ein akkumulierbares `UndoData`-Objekt, das den Status „abgeschlossen“ hat, gespeichert wird, wird für das nächste Objekt in jedem Fall ein neues Behälterobjekt erzeugt.
Um die für das Akkumulationskonzept eingeführten Statusvariablen in allen `UndoData`-Objekten abfragen und setzen zu können, wurde die `UndoData`-Schnittstelle um die folgenden vier Methoden erweitert:

```
public boolean canBeAccumulated()
public void setCanBeAccumulated(boolean canBeAccumulated)
public boolean isAccumulationCompleted()
public void setAccumulationCompleted(boolean accumulationCompleted)
```

Diese Methoden, sowie die beiden dazugehörigen Statusvariablen (**`boolean`** `_canBeAccumulated`, **`boolean`** `_accumulationCompleted`), sind in der Standardimplementierung bereits vorhanden. Die Statusvariablen sind beide mit den Werten `false` vorbelegt. Das bedeutet, daß die erste Variable für alle akkumulierbaren Aktionen geändert werden muß. Die zweite Variable muß nur geändert werden, wenn eine Akkumulation explizit abgeschlossen werden soll.
Im Integrationsserver wurde das Akkumulationskonzept eingesetzt, um für Texteingaben eine benutzerfreundliche Undo/Redo-Funktion zu realisieren. Die Eingabe jedes einzelnen Buchstabens in einem Textfeld ist eine eigenständige Aktion, die ein `UndoData`-Objekt erzeugt. Da im Integrationsserver für Texteingaben keine Bestätigungstaste verwendet wird, muß nach der Eingabe jedes einzelnen Buchstabens die vollständige Undo-Information in der `UndoData`-Liste vorliegen. Das Akkumulationskonzept ermöglicht es, Buchstaben z.B. wortweise zusammenzufassen. Dazu wird die Texteingabeaktion als akkumulierbar gekennzeichnet (`setCanBeAccumulated(true)`). Bei jedem vom Benutzer eingegebenen Leerzeichen wird zusätzlich das zugehörige `UndoData`-Objekt als abgeschlossen gekennzeichnet, bevor es gespeichert wird (`setAccumulationCompleted(true)`).

5.3.4 Nichteigenständige Aktionen

In den meisten größeren Anwendungen existieren Aktionen, die zwar als unabhängige Aktionen implementiert sind, aber für den Benutzer als solche nicht erkennbar sind. Diese Aktionen werden nur in Verbindung mit anderen ausgeführt und bekommen dabei keinen eigenen Eintrag in der Liste der Aktionen, die für die Undo- oder Redo-Funktion zur Verfügung stehen. Trotzdem müssen auch diese Aktionen berücksichtigt werden, wenn die Undo- oder Redo-Funktion ausgeführt wird. Nur so kann sichergestellt werden, daß sich die Anwendung wieder in exakt dem gleichen Zustand befindet, wie vor der Ausführung der Undo- oder Redo-Funktion.
Zu beachten ist, daß die nichteigenständigen Aktionen bei Undo zusammen mit der zuvor ausgeführten Aktion zurückgenommen werden, während sie bei Redo zusammen mit der nachfolgenden Aktion erneut ausgeführt werden. Ohne dieses asymmetrische Verhalten (siehe Abbildung 5.6) würde eine nichteigenständige Aktion entweder nach der Ausführung von Undo oder nach der Ausführung von Redo zu einem anderen Zeitpunkt als bei der ersten Ausführung auftreten.

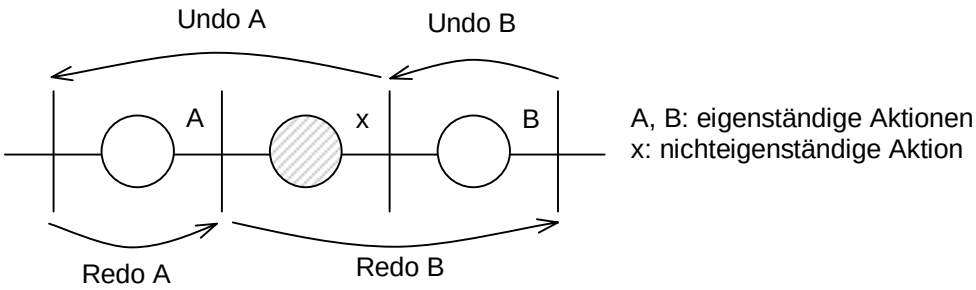

Abbildung 5.6: Undo und Redo von nichteigenständigen Aktionen

Ein Anwendungsfall für die Benutzung von nichteigenständigen Aktionen sind Darstellungsänderungen an graphischen Oberflächen, wie z.B. ein Wechsel der Sicht auf ein Material. Diese Darstellungsänderungen können entweder infolge einer anderen Aktion oder durch den Benutzer veranlaßt auftreten. Dabei wird in beiden Fällen kein Eintrag in der Befehlsgeschichte erzeugt, den der Benutzer für die Undo- oder Redo-Funktion auswählen kann. Trotzdem müssen diese Darstellungsänderungen zusammen mit der vorhergehenden bzw. nachfolgenden Aktion ausgeführt werden, wenn Undo bzw. Redo aufgerufen wird. Dies ist notwendig, weil sonst nach dem Aufruf von Undo oder Redo unter Umständen eine andere Sicht als bei der ersten Ausführung auf das Material gegeben ist.
Beispielsweise besteht im Integrationsserver die Möglichkeit, mit Hilfe von Reitern die Sicht zwischen verschiedenen Subwerkzeugen zu wechseln. Dies geschieht entweder auf Veranlassung des Benutzers oder bei der Ausführung von bestimmten Aktionen (z.B. Start der Integration). Der Wechsel der Reiter ist dabei als eine eigene Aktion implementiert, die aber nur zusammen mit einer weiteren Aktion zurückgenommen oder wiederholt wird.
Auch für die Kennzeichnung von nichteigenständigen Aktionen wird eine Statusvariable verwendet, die bei jedem `UndoData`-Objekt, das in der Befehlsgeschichte gespeichert werden soll, abgefragt wird. Anhand dieser Statusvariable wird in der Befehlsgeschichte entschieden, wie das Objekt gespeichert wird. Die dafür notwendige Methode, sowie die Methode mit der die Statusvariable verändert werden kann, werden ebenfalls in die `UndoData`-Schnittstelle aufgenommen:

```
public boolean isSeparateUndoStep()
public void setSeparateUndoStep(boolean isSeparateUndoStep)
```

In der Standardimplementierung (`UndoDataImpl`) ist diese Statusvariable (**`boolean`** `_isSeparateUndoStep`) mit `true` vorbelegt. Damit muß ihr Wert nur bei nichteigenständigen Aktionen verändert werden. Das heißt, ein neu erzeugtes `UndoData`-Objekt ist eigenständig, solange kein anderer Wert gesetzt wird. Auch Behälterobjekte (`CompoundUndoData`-Objekte) können als nichteigenständig markiert werden.
Die gleichzeitige Sicherung von eigenständigen und nichteigenständigen `UndoData`-Objekten erfordert einige Anpassungen innerhalb der `UndoData`-Liste, damit diese ohne Änderungen an ihrer externen Schnittstelle weiterbenutzt werden kann. Im einzelnen sehen ihre erweiterte Funktionalität und die dafür notwendigen Anpassungen der Methoden wie folgt aus:

Erweiterung der add()-Methode

Eigenständige und nichteigenständige `UndoData`-Objekte werden weiterhin in einer gemeinsamen internen Liste verwaltet. Um neue Objekte in die `UndoData`-Liste einzufügen, wird auch weiterhin nur eine Methode (`add(UndoData)`) benutzt. Diese in ihrer Implementierung erweiterte Methode behandelt die zu speichernden `UndoData`-Objekte in Abhängigkeit von ihren Eigenschaften unterschiedlich:

- Eigenständige `UndoData`-Objekte, die nicht akkumulierbar sind, werden direkt in der `UndoData`-Liste gespeichert (siehe Abbildung 5.7 a.).
- Eigenständige `UndoData`-Objekte, die akkumulierbar sind, werden
  - wenn dies möglich ist, zu einem bestehenden `CompoundUndoData`-Objekt hinzugefügt (siehe Abbildung 5.7 b.).
  - anderenfalls zu einem neu erzeugten `CompoundUndoData`-Objekt hinzugefügt und dieses in die interne Liste eingefügt (siehe Abbildung 5.7 c.).
- Nichteigenständige `UndoData`-Objekte werden
  - in einem neu erzeugten `CompoundUndoData`-Objekt temporär zwischengespeichert (also vorerst nicht in die `UndoData`-Liste eingefügt), falls dieses noch nicht existiert (siehe Abbildung 5.7 d.).
  - anderenfalls zu einem bestehenden zwischengespeicherten `CompoundUndoData`-Objekt hinzugefügt (siehe Abbildung 5.7 e.).
- Wenn zuvor ein oder mehrere nichteigenständige `UndoData`-Objekte zwischengespeichert wurden, zwischenzeitlich nicht Undo ausgeführt wurde und anschließend ein eigenständiges `UndoData`-Objekt gesichert werden soll, wird zuerst das `CompoundUndoData`-Objekt mit den nichteigenständigen und anschließend das eigenständige `UndoData`-Objekt gesichert (siehe Abbildung 5.7 f.).
  - Wenn nach der Sicherung zwei nichteigenständige `CompoundUndoData`-Objekte in der Befehlsgeschichte hintereinander stehen würden, wird das zweite Objekt zu dem ersten mit der `add()`-Methode hinzugefügt (siehe Abbildung 5.7 g.).

Da in der `UndoData`-Liste aufeinanderfolgende nichteigenständige `UndoData`-Objekte, unabhängig von ihrem Typ, immer zu einem `CompoundUndoData`-Objekt zusammengefaßt werden, steht in der internen Liste zwischen zwei eigenständigen höchstens ein nichteigenständiges `UndoData`-Objekt. Dies bringt erhebliche Erleichterungen bei der Implementierung der weiteren Methoden mit sich, da dadurch maximal ein Schritt weitergesprungen werden muß, um das nächste eigenständige `UndoData`-Objekt zu finden.
Weiterhin ist zu beachten, daß nichteigenständige `UndoData`-Objekte nicht als erste Elemente in der internen Liste gespeichert werden dürfen. Nichteigenständige Aktionen dürfen nur zusammen mit einer eigenständigen Aktion zurückgenommen werden, da sie für den Benutzer in der Befehlsgeschichte nicht sichtbar sind. Wenn aber ein `UndoData`-Objekt, das zu solch einer Aktion gehört, als erstes Element in der Liste steht, kann dieses keiner eigenstän-

digen Aktion zugeordnet werden. Deswegen speichert die `UndoData`-Liste nichteigenständige `UndoData`-Objekt gar nicht erst ab, solange sich der interne Zeiger am Anfang der Liste befindet.

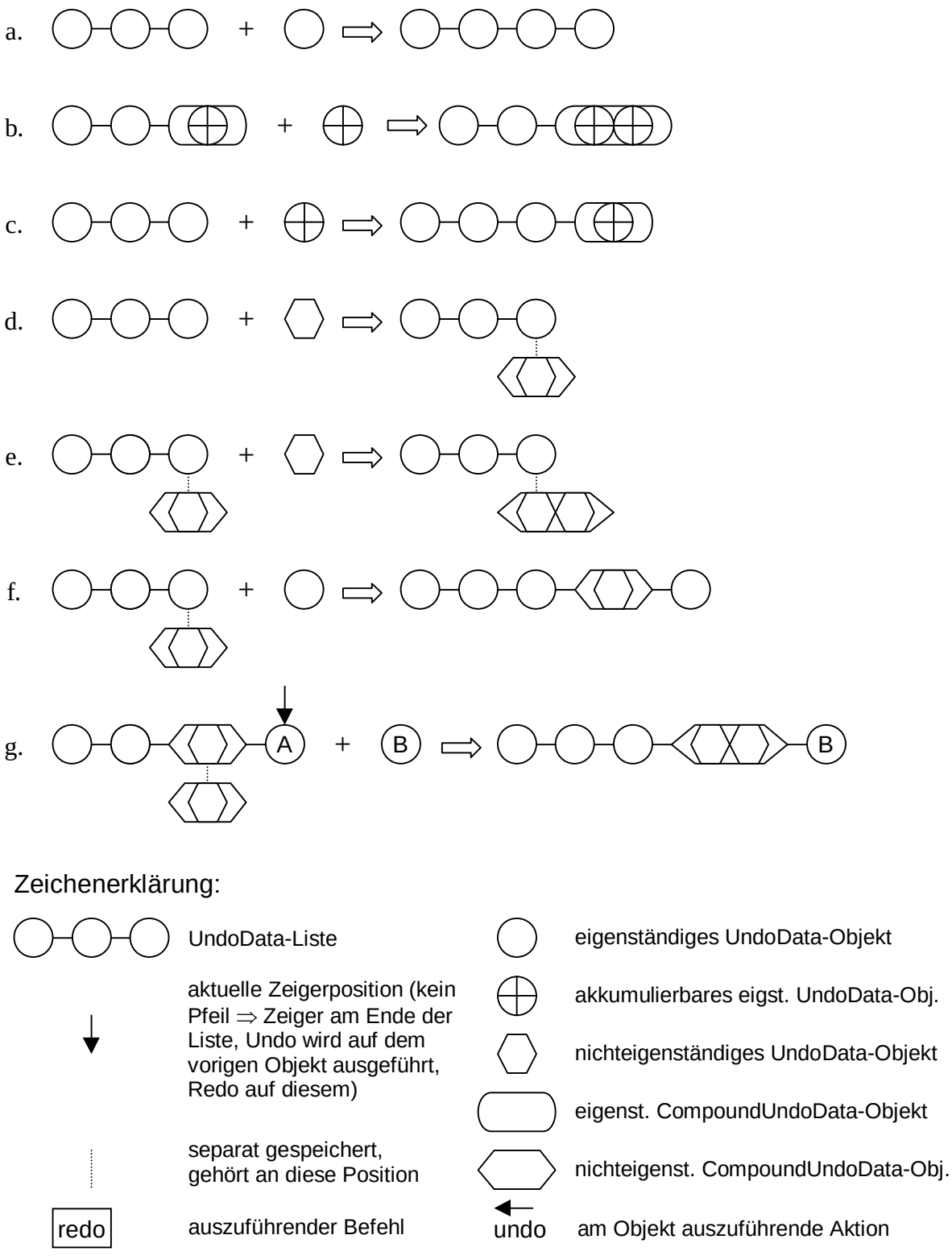

Abbildung 5.7: Beispiele für die Funktionsweise der erweiterten add()-Methode

Für das Ende der internen Liste gilt das gleiche, wie für den Anfang: Nichteigenständige `UndoData`-Objekte dürfen nicht an der letzten Position der Liste stehen, weil sie bei der Redo-Funktion keiner eigenständigen Aktion zugeordnet werden können. Wenn sich der interne Zeiger der Liste nicht am Ende befindet, würde zudem bereits durch die Ausführung einer nichteigenständigen Aktion die Befehlsgeschichte gelöscht und damit die Wiederherstellung der zuvor ausgeführten Aktionen unmöglich werden. Aktionen, die nur die Darstellung der Anwendung verändern, sollten aber keine direkten Auswirkungen auf die Befehlsgeschichte haben. Sobald aber eine eigenständige Aktion folgt, muß in beiden Fällen das nichteigenständige `UndoData`-Objekt an seiner ursprünglichen Position in der Liste gespeichert werden. Damit wird sichergestellt, daß sich die eigenständige Aktion bei einer späteren Benutzung von Undo oder Redo immer in dem Zustand befindet, den sie bei der ersten Ausführung inne hatte.
Gelöst wurde das Problem dadurch, daß nichteigenständige Aktionen zunächst immer in einem temporären Objekt zwischengespeichert werden. Erst wenn eine eigenständige Aktion folgt, werden das zwischengespeicherte und anschließend das eigenständige `UndoData`-Objekt in die `UndoData`-Liste eingefügt. Wird stattdessen vom Benutzer die Undo- oder Redo-Funktion aufgerufen, so wird die zwischengespeicherte nichteigenständige Aktion rückgängig gemacht, indem an ihrem `UndoData`-Objekt die `undo()`-Methode aufgerufen wird. Anschließend wird das temporäre Objekt gelöscht und die `undo()`-Methode am letzten bzw. die `redo()`-Methode am nächsten eigenständigen `UndoData`-Objekt der internen Liste aufgerufen (siehe Abbildung 5.8 a.). Mehrere aufeinanderfolgende nichteigenständige Aktionen werden auch in diesem Fall zusammengefaßt. Daher wird zur Speicherung der temporären Objekte immer ein `CompoundUndoData`-Objekt verwendet. Wenn in der internen Liste vor der Position, an der das temporäre `UndoData`-Objekt gespeichert werden soll, auch ein nichteigenständiges `UndoData`-Objekt steht, dann würden nach dem Einfügen zwei nichteigenständige Objekte hintereinander in der Liste stehen. Da dies nicht erlaubt ist, wird stattdessen das zu speichernde Objekt zu dem sich bereits in der Liste befindenden `CompoundUndoData`-Objekt mit der `add()`-Methode hinzugefügt.

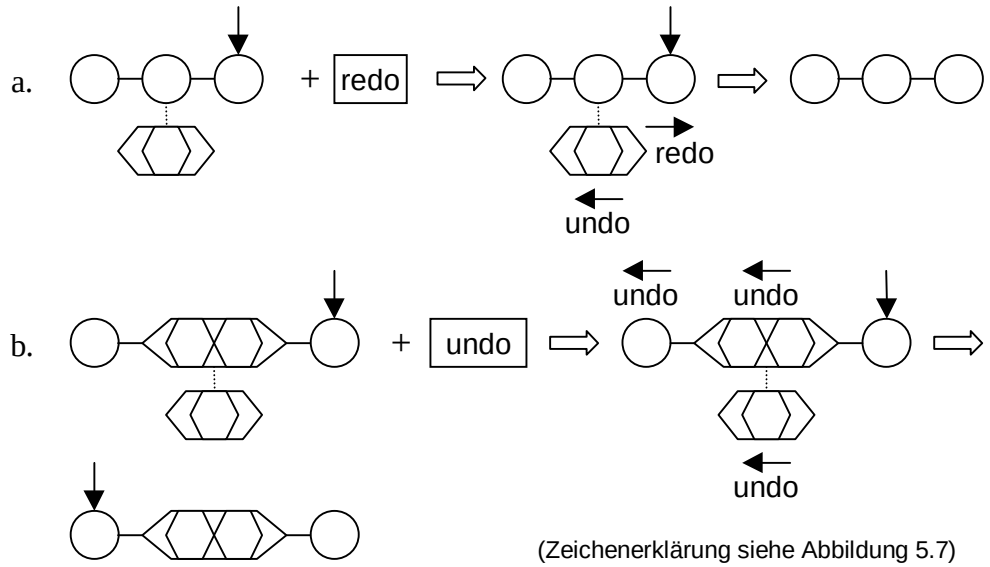

Abbildung 5.8: Beispiele für die Funktionsweise der undo()- und redo()-Funktionen

Änderungen an den Methoden getLast() und getNext()

Mit der `getLast()`- bzw. der `getNext()`-Methoden wird, von der aktuellen Zeigerposition aus betrachtet, das letzte bzw. das nächste `UndoData`-Objekt der `UndoData`-Liste zurückgegeben. Mit der Trennung in eigenständige und nichteigenständige Aktionen müssen die Implementierungen von `getLast()` und `getNext()` so geändert werden, daß jetzt nur noch die `UndoData`-Objekte eigenständiger Aktion zurückgegeben werden. Nichteigenständige Aktionen sollen von außerhalb der `UndoData`-Liste nicht mehr sichtbar und somit auch nicht mehr zugreifbar sein. Daher wird bei den neuen Implementierungen der beiden Methoden zunächst überprüft, ob das letzte bzw. das nächste `UndoData`-Objekt eigenständig ist. Sollte dies nicht der Fall sein, wird das jeweils davor bzw. danach kommende `UndoData`-Objekt zurückgeliefert. Daß es sich hierbei immer um ein eigenständiges Objekt handelt, wird dadurch sichergestellt, daß in der `UndoData`-Liste keine zwei nichteigenständigen Objekte hintereinander stehen dürfen und zudem das erste und das letzte Element der Liste eigenständig sein müssen.

Änderungen an den Methoden undoLast() und redoNext()

Die `undoLast()`- bzw. die `redoNext()`-Methode ruft in ihrer ursprünglichen Implementierung die `undo()`- bzw. die `redo()`-Methode am vorigen bzw. am nächsten `UndoData`-Objekt der `UndoData`-Liste auf. Mit der Einführung von nichteigenständigen Aktionen muß jetzt bei beiden Methoden zunächst überprüft werden, ob ein temporär zwischengespeichertes nichteigenständiges `UndoData`-Objekt existiert. Trifft dies zu, wird an diesem Objekt in beiden Fällen die `undo()`-Methode aufgerufen, um die zugehörige Aktion rückgängig zu machen. Anschließend wird, wie in der ursprünglichen Lösungsvariante, Undo bzw. Redo ausgeführt. Ist dabei das `UndoData`-Objekt, an dem `undo()` bzw. `redo()` aufgerufen wird, nicht eigenständig, wird auch noch am vorigen bzw. nächsten Objekt in der `UndoData`-Liste die jeweilige Methode aufgerufen (siehe Abbildung 5.8 b.).

Änderungen an den Methoden getHistoryForUndo() und getHistoryForRedo()

Mit der `getHistoryForUndo()`- bzw. der `getHistoryForRedo()`-Methode kann eine Liste der Namen aller Methoden, für die Undo bzw. Redo möglich ist, abgefragt werden. Mit der Einführung von nichteigenständigen Aktionen müssen die Implementierungen dieser Methoden so geändert werden, daß nur die eigenständigen Aktionen in die jeweilige Liste aufgenommen und zurückgegeben werden. Die nichteigenständigen Aktionen werden bei der Erstellung der Listen ignoriert.

5.4 Zusammenfassung

In diesem Kapitel wurde ein erweitertes Undo-Konzept vorgestellt, mit dem sowohl neu zu entwickelnde als auch bereits bestehende Anwendungen mit einer Undo-Funktion ausgestattet werden können. Das Konzept besteht aus einem Funktionskern und zusätzlichen Erweiterungen, die im Laufe der Entwicklung ergänzt wurden.
Mit Hilfe dieses Konzepts ließ sich der Integrationsserver mit relativ wenig Anpassungsaufwand um eine Unterstützung für Undo und Redo erweitern. Die Umstellung konnte dabei problemlos in mehreren Schritten erfolgen, ohne dabei zwischenzeitlich die Lauffähigkeit des Gesamtprogramms zu beeinträchtigen. An den Signaturen der Methoden, die für die Undo-Unterstützung erweitert wurden, erfolgten keine Änderungen. Daher konnten diese Methoden

jeweils durch eine neue Version mit Undo-Unterstützung ersetzt werden, ohne daß Anpassungen im übrigen Teil des Integrationsservers erforderlich waren.
Mit den zum Teil erst bei der Umstellung entwickelten zusätzlichen Konzepten kann auch für kompliziertere Anwendungen eine Undo-Unterstützung realisiert werden. Einzelne Konzepte und Lösungen konnten im Verlauf der Entwicklung direkt wiederverwendet werden: Zum Beispiel benutzen die durch die Einführung von nichteigenständigen Aktionen erweiterten Methoden der `UndoData`-Liste das Konzept der zusammenhängenden Aktionen, um die interne Struktur der `UndoData`-Liste übersichtlich zu halten. Aber auch Lösungen, die ursprünglich nur für einen speziellen Anwendungsfall gedacht waren, wurden in anderen Bereichen wiederverwendet: Das Akkumulationskonzept war ursprünglich nur dazu vorgesehen, für Texteingaben eine benutzerfreundliche Undo/Redo-Funktion zu schaffen. Dieses Konzept wurde später auch dazu verwendet, um eine größere Anzahl von Änderungen an der durch einen Vorschlag des Programms automatisch generierten Liste der zu integrierenden Dateien zusammenzufassen.
Die anfangs recht einfache und damit übersichtliche interne Struktur der `UndoData`-Liste wurde durch die Einführung der erweiterten Konzepte erheblich verkompliziert. Insbesondere die Unterscheidung zwischen eigenständigen und nichteigenständigen Aktionen machte es erforderlich, bei der Sicherung der Wiederherstellungsinformationen eine Vielzahl von unterschiedlichen Fällen zu behandeln.
Diese im Gegensatz zum ursprünglichen Konzept aufwendigere Verarbeitung innerhalb der `UndoData`-Liste sorgt dafür, daß der Benutzer auch bei zusammengesetzten Aktionen und Aktionen, die für ihn als solche nicht erkennbar sind, sein aus den meisten verbreiteten Standardanwendungen gewohntes Benutzungsmodell vorfindet.
Aus technischer Sicht gab es bei der Erweiterung des Integrationsservers um eine Undo-Funktion keine Probleme. Mit Hilfe des entwickelten Konzepts konnten die für eine Undo-Unterstützung notwendigen Anpassungen ohne große Schwierigkeiten vorgenommen werden. Die Problematik der Umstellung lag eher in dem benötigtem Detailwissen über die Anwendung. Bevor eine Methode um die Funktionalität ergänzt werden kann, mit der sie in den Zustand vor oder nach ihrer Ausführung zurückversetzt wird, muß zunächst genau bekannt sein, welche (Seiten-)Effekte durch ihre Ausführung auftreten. Bei einigen Methoden des Integrationsservers war dies zunächst nicht offensichtlich. Durch Seiteneffekte, die bei der Implementierung der Undo- und Redo-Methode anfangs nicht berücksichtigt wurden, geriet der Server in einen inkonsistenten Zustand. Erst nach einer genauen Analyse des Quellcodes konnten diese Fehler beseitigt werden.
Das Konzept ist ursprünglich mit dem Ziel entwickelt worden, in JWAM-Werkzeugen mit Trennung von Interaktions- und Funktionskomponenten einsetzbar zu sein. Da es sich bei monolithischen Werkzeugen um eine vereinfachte Form dieser Werkzeugkonstruktion handelt, ist das vorgestellte Konzept auch dort einsetzbar. Neben Werkzeugen dieser beiden Klassen wurde im Integrationsserverprojekt auch ein manipulatorbasiertes Werkzeug – der „Dependency Manager“ (siehe [Wei03]) – mit Hilfe dieses Konzepts um eine Undo/Redo-Funktion erweitert. Dafür waren ebenfalls nur geringe Anpassungen an dem Werkzeug notwendig. Das Undo-Konzept für IAK/FK-Werkzeuge ist damit wesentlich allgemeiner einsetzbar, als das im vorigen Kapitel vorgestellte Konzept für manipulatorbasierte Werkzeuge.

6 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein allgemeines Konzept zu entwickeln, mit dem auf dem JWAM-Rahmenwerk basierende Anwendungen mit relativ geringem Aufwand um eine Undo-Funktion ergänzt werden können. Dazu wurde zunächst in einem historischen Überblick die Entstehung und Verbreitung von Undo-Funktionen betrachtet. Anschließend wurden die wichtigsten Bestandteile und Eigenschaften von Undo-Funktionen beschrieben. Die hierbei genannten Anforderungen bildeten die Grundlage für die spätere Implementierung. Im letzten Teil von Kapitel 2 wurden verschiedene Benutzungsmodelle von Undo-Funktionen, die Undo-Modelle, vorgestellt und miteinander verglichen. Das Ziel dieses Vergleichs war es, ein Modell für die spätere Implementierung zu finden, das einen guten Kompromiß aus Funktionsumfang und Benutzbarkeit für den Anwender und Implementierungsaufwand für den Entwikkler darstellt. Die Wahl fiel hierbei auf das „eingeschränkte lineare Undo-Modell“. Neben den genannten Punkten hat es den Vorteil, das am häufigsten eingesetzte Undo-Modell zu sein. Die meisten Anwender dürften daher mit seiner Benutzung vertraut sein.
Das dritte Kapitel dieser Arbeit beschäftigt sich mit der praktischen Realisierung von Undo-Funktionen. Dabei wurden zunächst allgemeine Strategien zur Rekonstruktion eines früheren Anwendungszustands beschrieben. Anschließend wurde das Befehlsmuster und eine auf diesem basierende Undo-Funktion vorgestellt. Auf der Grundlage des Befehlsmusters bauen die meisten Undo-Funktionen von bekannten, sich im praktischen Einsatz befindlichen Anwendungen auf. Auch das im weiteren Verlauf dieses Kapitels vorgestellte allgemeine Undo-Konzept aus der Swing-Bibliothek von Java sowie die Undo-Funktion von JHotDraw bauen auf den Ideen des Befehlsmusters auf. Die Lösungsansätze dieser beiden Anwendungen wurden mit dem Ziel analysiert, herauszufinden, ob sich Teile von ihnen für das zu entwickelnde allgemeine Undo-Konzept weiterverwenden ließen. Während viele der Ideen übernommen werden konnten, mußte das Gesamtkonzept für die Verwendung mit JWAM-Anwendungen neu überarbeitet werden.
In Kapitel 4 wurde ein Undo-Konzept für manipulatorbasierte Anwendungen vorgestellt. Mit Hilfe einer Beispielanwendung wurden verschiedene Implementierungsvarianten einer Undo-Funktion, die auf diesem Konzept basiert, auf ihre Vor- und Nachteile untersucht. Als optimale Variante für die Beispielanwendung erwies sich dabei eine Lösung, die einen Kompromiß zwischen Implementierungsaufwand und Ressourcenverbrauch darstellt. Für Anwendungen, die auf Manipulatoren aufbauen oder eine ähnliche Architektur besitzen, kann mit diesem Konzept relativ einfach eine Undo-Funktion entwickelt werden. Bei anders aufgebauten Anwendungen muß dagegen erst aufwendig die Architektur angepaßt werden.
Daher wurde für die Entwicklung der Undo-Funktion des Integrationsservers ein anderes Konzept verwendet, das im fünften Kapitel dieser Arbeit vorgestellt wurde. Dieses Konzept hat den Vorteil, auch relativ einfach auf existierende Anwendungen übertragen werden zu können, ohne deren Architektur in großen Teilen ändern zu müssen. Die Erweiterung des Integrationsservers um eine Undo-Funktion erwies sich mit diesem Konzept als auf technischer Ebene problemlos realisierbar. Schwierigkeiten bereiteten dagegen fehlende Detailkenntnisse über die Implementierung der Anwendung.
Die beiden vorgestellten Undo-Konzepte sind für unterschiedliche Anwendungsfälle gedacht. Mit dem ersten Konzept können Undo/Redo-Funktionen für Anwendungen, die auf dem Manipulatorkonzept oder dem Befehlsmuster basieren oder die mit Hilfe dieser Konzepte realisiert werden sollen, entwickelt werden. Das zweite Konzept läßt sich unabhängig von der eingesetzten Art der Werkzeugkonstruktion verwenden und ist deshalb besser geeignet, wenn bestehende Anwendungen nachträglich um eine Undo-Funktion erweitert werden sollen.

Bei der Entwicklung des Integrationsservers erwies sich das Undo-Konzepts für IAK/FK-Werkzeuge als gut geeignet, um eine bestehende Anwendung um eine Undo-Funktion zu erweitern. Allerdings ist dieses Konzept nicht schlüsselfertig. Für jede Aktion, die mit einer Undo-Unterstützung versehen werden soll, müssen eine oder mehrere Methoden individuell angepaßt werden. Das vorgestellte Konzept liefert nur den Rahmen dafür, die Implementierungsdetails hängen von den jeweiligen Methoden ab. Eine universelle Undo-Funktion, die ohne individuelle Anpassungen eine Anwendung undo- und redofähig macht, ist – außer bei sehr kleinen Anwendungen – nicht zu realisieren.
Bei der nachträglichen Erweiterung einer existierenden Anwendung um eine Undo-Funktion treten insbesondere dann Probleme auf, wenn der Entwickler dieser Undo-Funktion nicht mit dem Quellcode der Anwendung vertraut ist. Gleichzeitig stellt die Undo-Unterstützung eine wichtige Hilfestellung für den Benutzer dar, die die Gebrauchstauglichkeit der Anwendung erheblich steigert. Daher ist zu überlegen, ob in Zukunft nicht viel mehr Anwendungen bereits bei ihrer Entwicklung mit einer Undo-Funktion ausgestattet werden sollten. Damit kann sich der zusätzlichen Aufwand erspart werden, der nötig ist, wenn eine Anwendung nachträglich um eine Undo-Funktion erweitert werden soll.

Literaturverzeichnis

[Abo91] Gregory D. Abowd, Alan J. Dix: *Giving undo attention*. In: Interacting with Computers Vol. 4, No. 3, 1991, pp. 317-342.

[Arc81] James E. Archer Jr., Richard Conway: *COPE: A cooperative programming environment.* Tech. Report TR81-459, Dept. of Computer Science, Cornell University, Ithaca, N.Y., Juni 1981.

[Arc84] James E. Archer Jr., Richard Conway, Fred B. Schneider: *User Recovery and Reversal in Interactive Systems.* In: ACM Transactions on Programming Languages and Systems, Vol. 6, No. 1, Januar 1984, pp. 1-19.

[Bec00] Kent Beck: *Extreme Programming Explained: Embrace Change*. Reading, Mass., etc.: Addison-Wesley, 2000.

[Ber93] Thomas Berlage: *Object-Oriented Application Frameworks for Graphical User Interfaces: The GINA Perspective*. GMD-Bericht. Nr. 213, München, Wien: R.Oldenbourg Verlag, 1993.

[Ber94] Thomas Berlage: *A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects*. In: ACM Transactions on Computer-Human Interaction, Vol. 1, No. 3, September 1994, pp. 269-294.

[Cho95] Rajiv Choudhary, Prasun Dewan: *A General Multi-User Undo/Redo-Model*. In: Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work, Stockholm, Sep. 10-14, 1995. Dordrecht: Kluwer Academic Publ., 1995, pp. 231-246.

[Dit03] Simon Ditrich: *Werkzeugkonstruktion mit Manipulatoren*. Diplomarbeit, Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, voraussichtlich 2003.

[Dix97] Alan Dix, Roberta Mancini, Stefano Levialdi: *The Cube – Extending Systems for Undo*. In: Design, Specification and Verification of Interactive Systems ’97: Proceedings of the European Workshop in Granada, Spain, June 4-6, 1997. Wien: Springer-Verlag, 1997, pp. 473-495.

[Eck98] Robert Eckstein, Marc Loy, Dave Wood: *Java Swing*. Beijing, etc.: O’Reilly, 1998.

[Ema02] The GNU Project: *GNU Emacs Manual*. http://www.gnu.org/manual/emacs-21.2/html_chapter/emacs.html

[Gam96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Bonn: Addison-Wesley, 1996.

[Gor85] Robert F. Gordon, George B. Leeman Jr., Clayton H. Lewis: *Concepts and Implications of Undo for Interactive Recovery*. In: Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective, Denver, Colo., Oct. 14-16, 1985. New York: ACM-Press, 1985, pp. 150-157.

[Gry96] Guido Gryczan: *Prozeßmuster zur Unterstützung kooperativer Tätigkeit*. Wiesbaden: Deutscher Universitäts-Verlag, 1996.

[Int02] IntelliJ: *IntelliJ IDEA-Homepage*. http://www.intellij.com/idea

[Jhd02] SourceForge: *JHotDraw Project-Page*, http://sourceforge.net/projects/jhotdraw/

[Joh92] Ralph E. Johnson: *Documenting frameworks using patterns*. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, Vancuver, Canada, Oct. 18-22, 1992. New York: ACM-Press, 1992, pp. 63-76.

[Jwa02] JWAM: *JWAM-Rahmenwerk-Homepage*. http://www.jwam.de

[Lee86] George B. Leeman Jr.: *A Formal Approach to Undo Operations in Programming Languages*. In: ACM Transactions on Programming Languages and Systems, Vol. 8, No. 1, Januar 1986, pp. 50-87

[Man96] Roberta Mancini, Alan Dix, Stefano Levialdi: *Reflections on Undo*. Technical Report RR9611, University of Huddersfield, 1996.

[Mes98] Tomer Meshorer: *Add an undo/redo function to your Java apps with Swing*. In: Java World Juni 1998, IDG Communications. http://www.javaworld.com/jw-06-1998/jw-06-undoredo.html

[Mey88] Bertrand Meyer: *Objektorientierte Softwareentwicklung*. München, London: Carl Hanser Verlag, Prentice-Hall, 1990.

[Mey97] Bertrand Meyer: *Object-Oriented Software Construction – Second Edition*. Upper Saddle River, N. J.: Prentice Hall, 1997.

[Mye96] Brad A. Myers, David S. Kosbie: *Reusable Hierarchical Command Objects*. In: Proceedings CHI'96: Conference on Human Factors in Computing Systems Vancouver, Canada. April 14-18, 1996. New York: ACM-Press, 1996, pp. 260-267.

[Ols98] Dan R. Olsen: *Developing User Interfaces*. San Francisco, Calif.: Morgan Kaufmann, 1998.

[Pra94] Atul Prakash, Michael J. Knister: A Framework for Undoing Actions in Collaborative Systems. In: ACM Transactions on Computer-Human Interaction, Vol. 1, No. 4, Dezember 1994, pp. 295-330.

[Ras00] Jef Raskin: *The Human Interface: New Directions for Designing Interactive Systems*. Reading, Mass.: Addison-Wesley, 2000.

[Rat87] Martin Rathke: *Dialogue Issues for Interactive Recovery - an Object-Oriented Framework*. In: Proceedings of the Second IFIP Conference on Human-Computer Interaction, INTERACT'87, Stuttgart, Sep. 1-4, 1987. Amsterdam: North-Holland, 1987, pp. 745-750.

[Res99] Matthias Ressel, Rul Gunzenhäuser: Reducing the Problems of Group Undo. In: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work, GROUP'99, Phoenix, Ariz., Nov. 14-17, 1999. New York: ACM-Press, 1999, pp. 131-139.

[Sch86] Kurt J. Schmucker: *Object-Oriented Programming for the Macintosh*. Hasbrouck Heights, N.J.: Hayden Book Company, 1986.

[Sch97] Hans-Jochen Schneider (Hrsg.): *Lexikon Informatik und Datenverarbeitung*. München: R. Oldenbourg Verlag, 1997.

[Sun02] JavaSoft: *Java 2 Platform, Standard Edition, v 1.4.0 API Specification, javax.swing.undo-Package*
http://java.sun.com/j2se/1.4/docs/api/javax/swing/undo/package-summary.html

[Thi90] Harold Thimbleby: *User Interface Design*. New York: ACM-Press, 1990.

[Tei78] Warren Teitelman: *Interlisp Reference Manual*. Palo Alto, Calif.: Xerox PARC, 1978.

[Tun01] Robert Tunkel: *Werkzeuge zur Unterstützung von Integrationsprozessen*. Studienarbeit, Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 2001.

[Uml01] Objekt Management Group: *OMG Unified Modeling Language Specification, Version 1.4*. September 2001.
http://www.omg.org/cgi-bin/doc?formal/01-09-67.pdf

[Vit84] Jeffrey S. Vitter: US & R: *A New Framework for Redoing*. In: ACM SIGPLAN Notices, Vol. 19, No. 5, 1984, pp. 168-176.

[Vli90] John M. Vlissides, Mark A. Linton: *Unidraw: A Framework for Building Domain-Specific Graphical Editors*. In: ACM Transactions on Information Systems, Vol. 8, No. 3, 1990, pp. 237-250.

[Wan91] Haiying Wang, Mark Green: *An Event-Object Recovery Model For Object-Oriented User Interfaces*. In: Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology, Hilton Head, S.C., Nov. 11-13, 1991. New York: ACM-Press, 1991, pp. 107-115.

[Wei03] Dennis Weise: *Visualisierung und Verwaltung von Abhängigkeiten zwischen Softwarekomponenten zur Compilezeit am Beispiel des JWAM-Frameworks*. Studienarbeit, Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, voraussichtlich 2003.

[Yan88] Yiya Yang: *Undo Support Models*. In: International Journal of Man-Machine Studies, Vol. 28, No. 5, 1988, pp. 457-481.

[Yan88a] Yiya Yang: *A new conceptual model for interactive user recovery and command reuse facilities*. In: Proceedings of CHI'88: Conference on Human Factors in Computing Systems,Washington D.C, May 15-19, 1988. Reading, Mass.: Addison-Wesley, 1988, pp. 165-170.

[Zül98] Heinz Züllighoven: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. Heidelberg: dpunkt.verlag, 1998.

Erklärung:

Ich versichere, daß ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Hamburg, den 22.01.2003

Björn Ostermann
Gärtnerstraße 105
20253 Hamburg
Matrikel-Nr.: 4822776