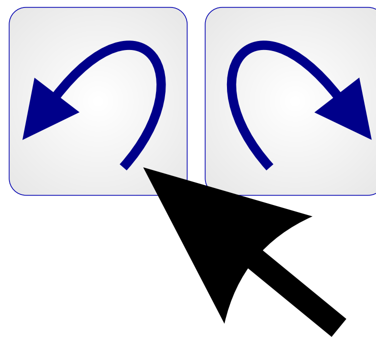
mbi
Technical Report
143 / 2003

Entwurf und Realisation eines generischen Interaktionsmodells mit Undo-Funktionalität für die medizinische Bildverarbeitung



Deutsches Krebsforschungszentrum Heidelberg
Abteilung Medizinische und Biologische Informatik

Ingmar Wegner

August 2003

Diese Arbeit wurde in der Abteilung Medizinische und Biologische Informatik (MBI), geleitet von Prof. Dr. H.-P. Meinzer, am Deutschen Krebsforschungszentrums (DKFZ) in Heidelberg durchgeführt und finanziert. Ich danke meinem Referenten, Herrn Prof. Dr. H.-P. Meinzer, für die Möglichkeit diese Arbeit anzufertigen. Ebenso danke ich Herrn Prof. Dipl.-Ing. Heinrich Krayl für die Übernahme des Korreferates. Mein weiterer Dank gebührt meinen Betreuern Dr. Ivo Wolf und Marcus Vetter, die mich mit gutem Rat und vielen produktiven Diskussionen begleitet haben. Vielen Dank auch an meine Kollegen der Abteilung MBI für das gute Arbeitsklima und die vielseitige Unterstützung.

Größter Dank gilt meinen Eltern, denn ohne die Motivation und finanzielle Unterstützung wäre das Studium nicht möglich gewesen.

Meiner Freundin Sylvie danke ich für die Hilfe und Interesse bei der Durchführung meiner Diplomarbeit.

Mit dem Abschluss des Studiums schaue ich zurück auf eine schöne und lehrreiche Zeit, die ich mit vielen Freunden teilen konnte. Jedoch blicke ich zugleich auch in eine interessante Zukunft.

Kurzfassung

Mit Hilfe der rechnergestützten medizinischen Bildverarbeitung können Ärzte z.B. besser analysieren und diagnostizieren, aber auch Operationen präziser planen und durchführen. Dabei wird häufig eine komplexe Mensch-Maschine-Interaktion benötigt, um das gewünschte Resultat zu erzielen. Fehler bei der Bedienung können nicht ausgeschlossen werden und so ist ein geschicktes Zusammenspiel zwischen Interaktion und Rücknahmemöglichkeit eines Befehls hilfreich.

Diese Diplomarbeit beschäftigt sich mit einer speziellen Methode, die Interaktion mit Rücknahmemöglichkeit so zu verwirklichen, dass sie schnell auf die Wünsche der Anwender angepasst werden kann, jedoch dem Entwickler alle nötigen Freiheiten lässt.

Schlagworte

Interaktion, Undo, Zustandsmaschine, MITK, VTK, ITK

Abstract

With the help of computer-aided medical image processing, physicians can analyse and diagnose more precisely, but also plan operations more accurately. In many cases a complex human-computer-interaction is required to gain the desired result. Because mistakes can not be avoided, a smart cooperation between interaction and undo is helpful.

This diploma thesis deals with a special method to realise the interaction with undo in a way, that it can be easily adapted to the user's wishes, but leaves all necessary freedom to the developer.

Keywords

interaction, undo, state machine, MITK, VTK, ITK,

Inhaltsverzeichnis

1 Einleitung 1

1.1 Motivation 1

1.2 Zielsetzung und Anforderungen 2

1.3 Aufbau und Konventionen 3

2 Grundlagen 5

2.1 Das Medical Imaging Toolkit 5

2.2 ITK 6

2.3 VTK 7

2.4 Generische Programmierung 7

2.5 Benutzerfreundlichkeit 8

2.6 Mensch-Maschine-Interaktion 9

2.7 Die Zustandsmaschine 10

2.7.1 Elemente 11

2.7.2 Stufen 12

2.7.3 Varianten 13

2.7.4 Zustandsübergangsdiagramm 13

2.7.5 Modellierung und Fehler 14

2.7.6 Wächter 16

2.7.7 Grundregeln für die Erstellung 18

2.7.8 Vor- und Nachteile 19

2.7.9 Testen 20

2.7.10 Software-Entwicklungen mit Zustandsmaschinen 20

2.7.11 Der Einsatz von Zustandsmaschinen 21

2.8 Undo/Redo 21

2.8.1 Benutzerfehler und ihre Revidierungsmöglichkeiten 22

2.8.2 Die Geschichte des Undos 22

2.8.3 Undo-Modelle 23

2.8.4 Anforderungen 28

2.8.5 Die Realisierung von Undo-Modellen 29

2.8.6 Die Wahl des Modells . . . 31

3 Entwurf **32**

3.1 Planung . . . 32

3.2 Entwurf des generischen Interaktionsmodells . . . 33

3.2.1 Zustandsmaschine . . . 35

3.2.2 Zustand und Übergang . . . 36

3.2.3 Generierung . . . 38

3.2.4 Definition . . . 39

3.2.5 Wiederverwendung der Interaktionsmuster . . . 40

3.2.6 Hierarchie . . . 41

3.2.7 Aktion . . . 43

3.2.8 Vererbung der Zustandsmaschinen . . . 44

3.2.9 Null-Übergang . . . 45

3.2.10 Umgang mit Ereignissen . . . 46

3.2.11 Zuordnung von Interaktions-Objekten zu Daten . . . 51

3.2.12 Gesamtübersicht des Interaktions-Modells . . . 51

3.3 Entwurf des Undo-Konzepts . . . 54

3.3.1 Kapselung der Operationen . . . 55

3.3.2 Operations-Klassen . . . 55

3.3.3 Speicherung in der Befehlsgeschichte . . . 56

3.3.4 Angebot von Undo-Modellen . . . 59

3.3.5 Eingeschränktes lineares Undo-Modell . . . 59

3.3.6 Granularität des Undos . . . 59

3.3.7 Gesamtübersicht des Undo-Konzepts . . . 61

4 Implementierung **63**

4.1 Konventionen im MITK . . . 63

4.2 Implementierung des Interaktions-Modells . . . 64

4.2.1 Zustandswechsel . . . 64

4.2.2 Generierung der Verhaltensmuster . . . 66

4.2.3 Test der Zustandsmaschine . . . 66

4.2.4 Wiederverwendung der Interaktionsmuster . . . 69

4.2.5 Beispiel für die Modellierung der Interaktion . . . 70

4.2.6 Anpassung, Erweiterung und Neuentwicklung . . . 72

4.3 Implementierung des Undo/Redo-Modells . . . 75

4.3.1 Eingeschränktes lineares Undo-Modell . . . 76

5 Ergebnis **78**

6 Diskussion und Ausblick **81**

Abbildungsverzeichnis **83**

Tabellenverzeichnis **85**

Glossar **85**

Literaturverzeichnis **89**

Index **95**

Kapitel 1

Einleitung

1.1 Motivation

Eine wichtige Aufgabe der medizinischen Bildverarbeitung ist die Unterstützung des medizinischen Entscheidungs- und Behandlungsprozesses.

Seit der Entdeckung der Röntgen-Strahlung (1895) ist der nicht-invasive Einblick in das Innere des menschlichen Körpers möglich. Die Durchleuchtung bietet jedoch nur eine zweidimensionale Ansicht auf die dreidimensionale Anatomie des Menschen. Im Laufe der Zeit wurden tomographische Bildgebungsverfahren entwickelt, die die Messung von morphologischen oder funktionalen Parametern in drei Dimensionen erlauben. Die Computertomographie (CT) oder die Positronen-Emissionstomographie (PET) erzeugen Schichtbildaufnahmen, wohingegen die Magnetresonanztomographie (MRT) sogar echte Volumendatensätze der anatomischen Strukturen erstellt. Durch die Verfahren konnte u.a. die Diagnostik präzisiert, aber auch die Operationsplanung beschleunigt werden. Ferner kann durch eine intraoperative Navigation die Qualität der Behandlung verbessert werden.

Die zunehmend präziser werdende Bildgebung führt jedoch auch zu immer größeren Datensätzen. Diese müssen durch geeignete Verfahren weiterverarbeitet und visualisiert werden, damit eine gute medizinische Behandlung möglich ist. Inzwischen unterstützen viele rechnergestützte Verfahren die Weiterverarbeitung der Datensätze. Für ein neues Verfahren wird oft auch eine neue Anwendung erstellt. Unnötigerweise wird der bereits für ein ähnliches Verfahren geschriebene Quellcode verworfen und neu implementiert. Die Neuentwicklungen beruhen u.a. auf den unterschiedlichen Anforderungen, die an ein Verfahren gestellt werden, und werden zudem durch Hardware- und Software-Abhängigkeiten verstärkt. Abhilfe schaffen Toolkits oder Frameworks, die für die Erstellung einer neuen Applikation die Wiederverwendung von bereits getestetem Quellcode anbieten. Beispiele sind das Visualisierungs-Toolkit *VTK* und das Segmentierungs- und Registrierungs-Toolkit *ITK*. Durch die Auswahl von Algorithmen und Modulen kann der Entwickler wertvolle Entwicklungszeit sparen. Weiterhin werden die verschmähten Doppelentwicklungen vermieden. Je nach Ziel der Entwicklung kann das erarbeitete Software-Paket in das Toolkit/Framework aufgenommen und für neue Entwicklungen bereitgestellt werden.

In vielen Entwicklungen ist eine Interaktion vorgesehen. Sie nimmt in der Bildverarbeitung einen besonderen Stellenwert ein und erlaubt dem Benutzer das Eingreifen in den Rechenvorgang. Dabei kann gerade bei medizinischen Anwendungen die Umsetzung dieser Manipulation zu einer komplexen Interaktion führen.

Mit zunehmender Komplexität der Interaktion werden Eingabefehler immer wahrscheinlicher. Besonders während einer Intervention muss die Möglichkeit der Rücknahme (Undo) einer Eingabe bestehen. Längst existieren Ansätze eines Undos, jedoch wird die Funktionalität selten in medizinischen Bildverarbeitungs-Anwendungen angeboten. Die meisten Vorhaben, eine Undo-Funktionalität im Nachhinein einzubinden, scheitern. Eine konsequente Undo-Funktionalität muss bereits vor Erstellung der Anwendung und parallel zu der Realisation der Interaktion geplant und realisiert werden. Eine zusätzliche Integration in ein Toolkit oder ein Framework ermöglicht eine umfassende Wiederverwendung.

1.2 Zielsetzung und Anforderungen

Ziel der Arbeit ist es, ein Interaktionskonzept zu entwerfen und zu realisieren, welches zusammen mit anderen Konzepten das *Medical Imaging Toolkit* (*MITK*) bildet. Das Interaktionskonzept soll neben einer flexiblen Modellierung auch die Möglichkeit eines Undos beinhalten.

Das Toolkit MITK soll dem Entwickler medizinischer Bildverarbeitungs-Anwendungen als Grundgerüst für die Erstellung wissenschaftlicher Arbeiten dienen. So muss nicht erst eine unter Umständen komplexe Entwicklungsumgebung geschaffen werden, die u.a. Visualisierung, Interaktion oder Datenhaltung umfasst. Stattdessen kann auf bereits implementierte Komponenten zurückgegriffen werden.

Das MITK befindet sich seit November 2002 im Aufbau. Die Entwicklung des Interaktions- und des Undo-Konzepts soll sich in die Entwicklung der übrigen Komponenten integrieren.

Für die Modellierung des *Graphical User Interface* (*GUI*) wird das Framework *QT* [Tro03b] [Dah99] von Trolltech genutzt. Dabei wird jedoch, so weit es geht, unabhängig von QT entwickelt, um jederzeit eine Verwendung eines anderen GUI-Software-Pakets zu gewährleisten. So weit möglich werden die Visualisierung über das Toolkit VTK, die Segmentierung sowie Registrierung über das Toolkit ITK realisiert.

Da es sich um ein allgemeines medizinisches Toolkit handelt, müssen die Konzepte eine maximale Flexibilität erlauben. Ferner sollen sie sich so benutzerfreundlich wie möglich dem Anwender sowie dem Entwickler zeigen. Zu achten ist auf Übersichtlichkeit und Wartbarkeit.

Wünschenswert ist eine Möglichkeit der schnellen Änderung der Interaktion sowie eine Bereitschaft, unterschiedliche Interaktions-Schemata für unterschiedliche Benutzer zu erstellen. Die Unterschiede zeigen sich meist im Detail, z.B. ob nach dem Anklicken eines

neuen Objekts nur das vorher markierte Objekt deselektiert wird oder ob zusätzlich das neue Objekt mit selektiert wird. Es handelt sich hierbei zwar oft nur um Feinheiten, aber das sind gerade die Aspekte, die eine Software benutzerfreundlich machen.

Unbedingt jedoch muss die Interaktion schnell neuen Begebenheiten gegenüber anpassungsfähig sein. Diese Begebenheiten schließen neue Eingabegeräte oder neue Bildverarbeitungsmethoden mit ein. Das Interaktionskonzept soll besonders vorteilhaft für die Realisation einer komplexen Interaktion sein. Für die schnelle Erstellung von Anwendungen soll es ermöglicht werden, Standardinteraktionen einzusetzen.

Das Undo-Konzept wird sich in die Interaktion nahtlos integrieren. Es soll ebenso multifunktionell wie flexibel sein und alle Anwendungsfälle unterstützen. Dabei ist besonders auch auf Benutzerfreundlichkeit zu achten.

Zwei Beispiele geben eine Vorstellung über den Einsatz einer medizinischen Interaktion: *Aktive Konturen* [Kun00] helfen bei der zweidimensionalen *Segmentierung* von Organen in einem dreidimensionalen Datensatz. Der Anwender umfährt die anatomische Struktur, an der er interessiert ist, mit der Maus oder mit einem Stift auf einem Grafiktablett. Die aktiven Konturen legen sich nun an die durch bestimmte Merkmale charakterisierte Form an. Falls der Anwender mit dem Ergebnis nicht zufrieden ist, kann er es durch einen virtuellen Radiergummi oder andere Substraktionswerkzeuge verändern.

Das zweite Beispiel ist ein interaktives Programm, welches für die *Navigation* von medizinischen Instrumenten im Körperinneren notwendig ist. Ein Datensatz wird *präoperativ* beispielsweise durch Computertomographie (CT) aufgenommen und zur Operationsplanung verwendet. Will der Spezialist während der Operation eine Ansicht auf diese Planungsdaten und dem medizinischen Instrument haben, so werden weitere *intraoperative* CT-Aufnahmen benötigt. Erst durch diese kann eine Registrierung der Planungsdaten und der intraoperativen Daten durchgeführt werden. Eine interaktive Anwendungssoftware übernimmt den Abgleich. Beide Datensätze werden überlagert auf einem Bildschirm angezeigt. Der medizinische Spezialist setzt in beiden Datensätzen Korrespondenzen, die der Anwendung als Anhaltspunkt für die Berechnung der Ansicht dienen. Mehrere Punktepaare müssen gesetzt und bei Bedarf auch verschoben oder gelöscht werden können.

Die Funktionalität, Punkte zu setzen, zu verschieben und auch zu löschen, wird von den meisten Verfahren benötigt werden. Um das Interaktionskonzept zusammen mit dem Undo-Konzept zu präsentieren, soll diese Interaktion realisiert werden. Mit Vollendung der Arbeit soll es möglich sein, interaktive Anwendungen mit dem MITK zu erstellen.

1.3 Aufbau und Konventionen

Im Folgenden wird zunächst der Aufbau der einzelnen Kapitel dieser Arbeit beschrieben. Die Kapitel bauen zwar aufeinander auf, jedoch kann je nach Wissensstand oder auch Interesse

das eine oder andere Kapitel ausgelassen werden. Die Konventionen, nach denen sich diese Arbeit richtet, werden im Anschluss aufgelistet.

In *Kapitel 2* werden alle nötigen Grundlagen dieser Arbeit besprochen. Besonderes Augenmerk wird auf die Theorie und Praxis der Zustandsmaschine gelegt. Ebenso genau wird auf die verschiedenen Möglichkeiten des Undos/Redos von Befehlen eingegangen.

Kapitel 3 beschäftigt sich mit dem Entwurf der Konzepte Interaktion und Undo/Redo. In diesem Kapitel wird sehr genau auf die einzelnen Ideen, die den Konzepten zugrunde liegen, eingegangen. Ferner wird ein kurzer Überblick über die Durchführung der Planungsphase geboten.

In *Kapitel 4* werden die Ideen anhand des implementierten Quellcodes noch näher beschrieben. Dabei werden kurze Auszüge des Codes mit Fokus auf Bedeutsamkeit besprochen.

Kapitel 5 beschreibt das Ergebnis dieser Arbeit und bietet gleichzeitig eine Zusammenfassung.

In *Kapitel 6* wird das Ergebnis dieser Arbeit kritisch hinterfragt und die Aussichten des Ergebnisses besprochen. Ferner werden Vorschläge zur Erweiterung genannt.

In einem Index werden zentrale Begriffe aufgeführt und in einem Glossar, wenn nötig, näher erläutert.

Grafische, sprachliche und formale Konventionen

Wichtige Begriffe werden bei erster Verwendung *kursiv* geschrieben. Zur Hervorhebung und Unterscheidung werden seltene Begriffe wiederholt kursiv geschrieben. Einige Begriffe, für die es keine oder nur künstlich wirkende Übersetzungen in der deutschen Sprache gibt, werden nicht übersetzt. Beispiele hierfür sind Undo oder Redo.

In dieser Arbeit wird auf verwirrende Synonyme verzichtet, da exakte Begriffe die Eindeutigkeit fördern. Auf Grund dessen werden manche Begriffe als zu häufig genutzt erscheinen. Da es sich jedoch um zentrale Begriffe handelt, ist dies unumgänglich.

Jeglicher Programmcode, d.h. Klassen- und Objekt-Namen, Variablen etc., werden mit der Schriftart `Verbatim` gekennzeichnet. Ferner ist der Quellcode gemäß der Konventionen des MITKs in englischer Sprache geschrieben.

Zustands-Diagramme sind überwiegend in traditioneller Notation gezeichnet. In Ausnahmefällen wird die zugrunde liegende Notation näher beschrieben. Klassen- sowie Objekt-Diagramme sind nach UML-Standard gezeichnet.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit besprochen. Zu Beginn werden das Toolkit *MITK* als Entwicklungsumfeld der Arbeit und die hierzu verwendeten Toolkits *ITK* und *VTK* beschrieben. Eine kurze Beschreibung über generische Programmierung folgt darauf.

Um die Notwendigkeit von später vorgestellten Mechanismen zu erklären, wird im Anschluss ein kurzer Überblick über die Benutzerfreundlichkeit von Programmen geboten und die Problematik der Mensch-Maschine-Interaktion beschrieben. Hiernach folgen die Kapitel Zustandsmaschinen und Undo/Redo, die in dieser Arbeit eine übergeordnete Rolle spielen und somit detaillierter ausfallen.

2.1 Das Medical Imaging Toolkit

Das im November 2002 initiierte *Medical Imaging Toolkit* (MITK) unterstützt die Entwicklung von interaktiven medizinischen Bildverarbeitungsprogrammen mit Bedarf an komplexer Interaktion. Es vereint die Segmentierungs- und Registrierungs-Algorithmen von ITK (vgl. 2.2) mit den Visualisierungs-Algorithmen von VTK (vgl. 2.3) und fügt u.a. folgende Funktionalitäten hinzu:

- Vereinfachung und Konsistenz multipler Ansichten auf einem Datensatz
- Veränderung eines Datensatzes mittels Interaktion
- Realisation komplexer Interaktion durch hierarchische Zustandsdefinition
- Undo/Redo-Funktionalität
- semantische Organisation der Daten in einem Datenbaum
- Visualisierung und Interaktion von zeitaufgelösten dreidimensionalen Datensätzen (3D+t)
- einheitliche Beschreibung von 3D+t Datensätzen

Nach der Entwicklungs- und grundlegenden Implementierungsphase wird MITK allen interessierten Entwicklern zur Verfügung gestellt (*Open-Source-Projekt*). Dabei profitiert MITK von den Vorteilen eines Frameworks [S+95].

Der Quellcode des MITKs richtet sich nach den Grundsätzen der generischen Programmierung (vgl. 2.4) und ist plattformunabhängig. Er wird regelmäßig durch ein Qualitätsmanagement überprüft (Dart-Tool, Dashboard): Das viel benutzte Versions Management System „Concurrent Versions System“ (CVS) [Ced02] ermöglicht paralleles Arbeiten der MITK-Entwickler. Nach der Fertigstellung eines Abschnitts übermittelt der Entwickler seine Arbeit an einen Server. Auf diesem wird eine Nachricht an ein so genanntes „Dart-Tool“ geschickt. Dieses Tool überprüft den gesamten Quellcode auf unterschiedlichen Betriebssystemen zu unterschiedlichen Bedingungen und erstellt das Ergebnis in Form einer HTML-Webseite („Dashboard“). Der Entwickler überprüft das Ergebnis und behebt gegebenenfalls die Fehler.

Das Ziel des Medical Imaging Toolkits ist es, den Nutzen von ITK und VTK zu vereinen und sie durch die Realisation komplexer Interaktionsmechanismen zu ergänzen. Dadurch wird ermöglicht, Anwendungen der interaktiven medizinischen Bildverarbeitung schnell und ohne Doppelentwicklung zu realisieren.

2.2 ITK

1999 gründete Dr. Terry Yoo in den USA die Arbeitsgemeinschaft „Insight Toolkit“ (ITK), die aus drei kommerziellen und weiteren drei akademischen Organisationen besteht. Zweck des Zusammenschlusses ist die Erstellung eines Open-Source-Projekts für die Registrierung und Segmentierung im medizinischen Bereich. Der zur freien Verfügung gestellte Quellcode ist unabhängig vom Betriebssystem, unterstützt *Multithreading* und richtet sich nach den Grundsätzen der generischen Programmierung (vgl. 2.4). Ein besonderes Konzept ist der Einsatz von so genannten Smart-Pointern, ein Mechanismus, der die Anzahl von Referenzen auf ein Objekt hält und den belegten Speicherplatz des Objekts automatisch freigibt, wenn die Anzahl unter eins fällt.

Die Ziele des Projekts erstrecken sich über:

- Unterstützung des Visible Human Projekts
- Gründung einer Stiftung für zukunftsorientierte Forschung
- Aufbauen einer Bibliothek aus grundlegenden Algorithmen
- Entwicklung einer Plattform für fortgeschrittene Produktentwicklung
- Erstellung von Konventionen für zukünftige Arbeiten

- Gründung einer selbsterhaltenden Gemeinschaft aus Software–Benutzern und –Entwicklern

Für nähere Informationen verweise ich hier auf die ITK-Homepage: http://www.itk.org

2.3 VTK

Das „Visualisation Toolkit“ (VTK) ist wie ITK ein Open-Source-Projekt, aber mit dem Unterschied, dass VTK eine Bibliothek zur 3D-Computergrafik, Bildverarbeitung und Visualisierung zur Verfügung stellt. In nur wenigen Anweisungen kann eine komplette Visualisierung implementiert werden.

VTK wird von vielen kommerziellen Software-Entwicklungen genutzt. Um hierfür die Qualität des Quellcodes zu gewährleisten, werden Mechanismen öffentlich bereitgestellt, die erlauben, die Konsistenz des gesamten Quellcodes zu überprüfen. So wird CVS eingesetzt, damit die Vielzahl der Entwickler parallel arbeiten können. Nach der Fertigstellung eines Abschnitts übermittelt der Entwickler seine Arbeit an den VTK-internen Server. Dieser teilt die Änderung dem „Dart-Tool“ mit, welches dann den gesamten Quellcode zu unterschiedlichen Bedingungen überprüft. Der Entwickler kann das Ergebnis nach Überprüfung in einer auf der VTK-Homepage zur Verfügung gestellten Tabelle („Dashboard“) einsehen und bei Fehlern diese dann verbessern.

Das Toolkit VTK umfasst mittlerweile viele Algorithmen, die ständig weiterentwickelt werden. Der Quellcode ist betriebssystemübergreifend, unabhängig von GUI-Umgebungen und unterstützt *Multithreading*.

Weitere Informationen auf der VTK-Homepage: http://public.kitware.com/VTK

2.4 Generische Programmierung

Früher glaubte man, dass die Wiederverwendbarkeit von Objekten ein Nebenprodukt der objektorientierten Programmierung sei [HC91]. Der Quellcode, der bei einem großen Software-Projekt geschrieben wurde, könne so für ein weiteres Projekt genutzt werden. Aber das hat sich im Laufe der Zeit als falsch herausgestellt [CE00, S. 60]. Die implementierten Objekte waren häufig zu sehr auf eine Aufgabe spezialisiert oder an einen Typ (int, bool etc.) gebunden, um sie für eine weitere Implementierung in einem anderen Zusammenhang zu nutzen. Um wirkliche Wiederverwendbarkeit zu erreichen, musste Mehrarbeit geleistet werden, die kaum mit dem Budget eines Software-Projekts vereinbar war. *Generische Programmierung* beseitigt dieses Problem, denn hier konzentriert sich der Entwickler nicht auf eine spezielle Aufgabe oder einen Datentyp, sondern auf eine ganze Familie von Aufgaben bzw. Datentypen.

Das einfachste Beispiel ist die Implementierung durch *Templates*. Sie gruppieren alle Arten von Datentypen in eine Familie. Der Entwickler nutzt in einer Methode keine speziellen Typen, sondern arbeitet mit der Familie von Typen. Später nimmt ihm der Computer die Aufgabe, den Platzhalter Template durch den speziellen Typ zu ersetzen, ab. Die Mehrarbeit übernimmt somit der Computer.

Als ein umfangreiches Nachschlagewerk gilt „Generative Programming“ von Krzysztof Czarnecki und Ulrich W. Eisenecker [CE00].

2.5 Benutzerfreundlichkeit

Mit der *Benutzerfreundlichkeit* eines Programms steigt und fällt der Erfolg einer Software-Entwicklung. Um eine möglichst hohe Benutzerfreundlichkeit zu erreichen, gehen wir im Folgenden auf die Grundprinzipien ein.

In ISO 9241, Teil 11, wird Benutzerfreundlichkeit (usability) wie folgt beschrieben:

> „The extent to which a product (e.g. tool) can be used by specified users to achieve specified goals with effectivity, efficiency and satisfaction in a specified context of use.“

Quelle: ISO 9241, part 11

Laut National Cancer Institut in den USA wird Benutzerfreundlichkeit folgendermaßen definiert:

> Usability ist das Maß für die Qualität eines Benutzererlebnisses bei der Interaktion mit einem Produkt oder System.

Quelle: NCI

Dieses Benutzererlebnis wird durch unterschiedliche Faktoren beeinflusst. Zum einen spielt die Einfachheit des Erlernens eine große Rolle, zum anderen erleichtern die Fähigkeit der Einprägung und die Effizienz die Benutzung. Hohe Fehlerhäufigkeit oder große Tragweite der Fehler können der Benutzerfreundlichkeit erheblich schaden. Sie wird jedoch auch durch die subjektive Zufriedenheit des Benutzers beeinflusst.

Für die Erstellung von Benutzerschnittstellen wurden die so genannten *acht goldenen Regeln* (nach DIN EN ISO 9241) verfasst:

1. Konsistenz
2. Shortcuts (z.B. Ctrl-C)
3. Informatives Feedback

4. Dialoge mit definierten Abschlüssen

5. Fehlervermeidung, einfache Fehlerbehandlung

6. Rücknahmemöglichkeit (Undo)

7. Vorhersehbare Systemreaktionen

8. Keine Überforderung des Kurzzeitgedächtnisses

Punkt Nummer sechs verweist auf eine Undo-Funktionalität. Eine 1980 von Card et al. durchgeführte Studie ergab, dass 31% aller erfahrenen Benutzer Fehler oder ineffektive Eingaben bei der Erledigung von Aufgaben mit Texteditor oder Betriebssystem begehen [CMN80]. Eine weitere Studie, Brown und Gould (1987), ergab, dass ca. 50% aller erfahrenen Benutzer Fehler beim Erstellen von Tabellen unterlaufen [BG87].

Die Studien belegen den Punkt *Rücknahmemöglichkeit* und zeigen, dass ein Undo-Mechanismus, wie in 2.8 beschrieben, die Benutzerfreundlichkeit besonders im medizinischen Umfeld um ein Vielfaches steigert.

2.6 Mensch-Maschine-Interaktion

Um das Einsatzgebiet des MITKs kennen zu lernen, folgt ein kurzer Überblick über Interaktion mit Vorschlägen zum Nachlesen. Gehen wir zunächst auf die allgemeine Definition von Interaktion ein:

> „Aufeinander bezogenes Handeln zweier oder mehrerer Personen; Wechselbeziehung zwischen Handlungspartnern (Psychol.; Soziol.)“

Quelle: Duden Fremdwörterbuch

Die Mensch-Maschine-Interaktion bezieht sich auf eine Wechselwirkung zwischen Mensch und Computer. Schmidt beschreibt die Mensch-Maschine-Interaktion folgendermaßen:

> „...the user tells the computer in a certain level of abstraction (e.g. by command-line, direct manipulation using a GUI, gesture, or speech input) what she expects the computer to do.“

Quelle: [Sch00, S. 2]

Stark et al. beschreiben die Mensch-Maschine-Interaktion mittels Handgesten [SKZ95] und Arons schildert eine rein sprachbasierte Interaktion [Aro91].

Beschreiben wir die Problematik an einem konkreten Fall. Ein Aufgabengebiet des MITKs ist die Segmentierung, in der der Benutzer, hier der medizinische Experte, ein für ihn interessantes Gebiet (region of interest (ROI)) aus einem Datensatz von Schichtaufnahmen auswählen will. Ihm stehen für die Segmentierung eine Anzahl von Werkzeugen zur Verfügung, mit denen er in den Ansichten des Datensatzes arbeiten kann. Die Werkzeuge untergliedern sich in automatische, semiautomatische und manuelle Segmentierungswerkzeuge. Jedes einzelne Werkzeug besitzt ein spezielles Verhalten und benötigt deshalb auch einen speziellen Umgang. Ein Werkzeug kann Parameter benötigen, die vorher definiert werden müssen, oder kann nur in einer dreidimensionalen Ansicht seine Aufgabe erfüllen. Ein Freihandzeichnen-Werkzeug ist nicht sehr komplex und verhält sich wie ein Stift auf Papier. Ein Polygon definiert ein Gebiet (ROI) durch Stützstellen (Punkte). Ein *region-growing*-Werkzeug selektiert in einem Datensatz von einem Saatpunkt aus Pixel oder Voxel, deren Grauwertbereich zuvor eingestellt oder über sonstige Eingaben spezifiziert wurde [Hei03]. Andere Werkzeuge fordern eine noch komplexere Interaktion und werden ständig weiterentwickelt.

Das Problem, um welches sich die Interaktion kümmert, ist, dem Benutzer eine einfache Möglichkeit zu bieten, die eigenen Vorstellungen über Manipulation und Ansicht dem Computer mitzuteilen.

Hilfreich wäre ein Interaktionskonzept, bei dem alle Werkzeuge, egal wie komplex, intuitiv zu gebrauchen sind, d.h. ein ungeübter Benutzer ohne Erlernen sofort damit arbeiten könnte. So ist die Gefahr geringer, dass sich der Experte auf eine kleine Auswahl an Werkzeugen spezialisiert und andere, besser geeignete Werkzeuge außer Acht lässt.

Weitere Informationen über die interaktive Segmentierung bietet Kunert in [Kun00] und Olabarriaga in [Ola99] umfassend und übersichtlich an. Preece beschreibt die Mensch-Maschine-Interaktion in [Pre94] näher. Den Bezug zur Software-Entwicklung stellt Macauly in [Mac95] her.

2.7 Die Zustandsmaschine

Das Konzept der *Zustandsmaschine* (state machine) nimmt in dieser Arbeit eine zentrale Rolle ein. Im Folgenden wird das Konzept detaillierter beschrieben.

Zustandsmaschinen begleiten uns den ganzen Tag über. Sie stecken in Kaffeemaschinen, Ticketautomaten, Fernsehgeräten, Uhren, Stereoanlagen und vielen weiteren Geräten. Auch in Betriebssystemen und Computer-Anwendungen sind sie präsent. Auf einer abstrakten Ebene können sie durch drei Charakteristika identifiziert werden:

- Sie besitzen nur eine endliche (finite) Anzahl von Zuständen.

- Die Ausgabe ist abhängig von der Eingabe und dem aktuellen Zustand.
- Sie gehen von dem aktuellen Zustand nur durch eine Benutzereingabe oder einer eigens erzeugten Eingabe in den nächsten über.

Systeme mit diesen Eigenschaften werden formal als *finite Zustandsmaschinen* bezeichnet. *Deterministische Zustandsmaschinen* erfüllen die Anforderung, dass in ihnen kein Zustand existiert, in dem Übergänge definiert sind, die bei gleichem Ereignis in zwei unterschiedliche Zustände führen (vgl. Abb. 2.6).

Ein Beispiel für eine typische Zustandsmaschine ist das Kombinationsschloss (vgl. Abb. 2.1). Während bei einem gewöhnlichen Zahlenschloss die Reihenfolge der Eingabe nicht entscheidend ist, ist bei einem Kombinationsschloss, wie der Name schon sagt, die Kombination der Eingaben, hier Zahlen, von Bedeutung. Gibt man die richtigen Zahlen in falscher Reihenfolge ein, so akzeptiert das Kombinationsschloss die Eingabe nicht und bleibt geschlossen, wohingegen das Zahlenschloss bei jeder Reihenfolge öffnet.

Abbildung 2.1: *links*: Zahlenschloss; *rechts*: Kombinationsschloss an einem victorianischen Safe

2.7.1 Elemente

Abbildung 2.2: Die vier Elemente einer Zustandsmaschine (traditionell): Zustand mit Nummer (rot), Übergang (schwarz), Ereignis (grün) und Aktion (blau)

Eine Zustandsmaschine setzt sich aus folgenden Elementen zusammen (vgl. Abb. 2.2):

- Zustand (State): ein Konstrukt, welches die Information der bisherigen Ereignisse verkörpert

- Übergang (Transition): wird durch ein bestimmtes Ereignis ausgelöst und führt von einem akzeptierenden Zustand in einen resultierenden Zustand
- Ereignis (Event): eine Eingabe oder ein zeitliches Intervall
- Aktion: das Resultat, das auf ein Ereignis folgen kann

Eine Zustandsmaschine, die als Unterklasse von Petri-Netzen [Rei85] gilt, kann aus beliebig vielen der oben genannten Elementen zusammengesetzt sein. Jedoch hat jede Zustandsmaschine genau einen *initialen* Zustand, von dem aus das System beim ersten Ereignis startet.

Jedem Übergang ist ein akzeptierender und ein resultierender Zustand zugeordnet. Hierbei kann es sich auch um ein und denselben Zustand handeln. Es werden nur Ereignisse akzeptiert, die in der Maschine definiert sind. Andere, eventuell neu hinzugekommene, werden wie nicht definierte Ereignisse ignoriert. Eine Maschine befindet sich immer nur in einen Zustand, dem *aktuellen* oder *aktiven* Zustand.

Eine Zustandsmaschine kann beliebig viele *finale* Zustände beinhalten. In einem solchen Zustand beendet die Maschine ihre Tätigkeit und beachtet keine weiteren Ereignisse mehr. Eine weniger restriktive Definition von finalen Zuständen findet man in formalen Modellen: Ein finaler Zustand ist der Zustand, der erreicht wird, nachdem eine komplette Folge von Ereignissen akzeptiert worden ist. In diesem Zustand akzeptiert die Maschine weitere Ereignisse, es werden jedoch keine weiteren erwartet.

Eine Zustandsmaschine ist streng statisch. Sie darf während des Betriebs nicht verändert werden, d.h. es dürfen keine Zustände, Übergänge, Ereignisse oder Aktionen hinzugefügt, verändert oder entfernt werden.

2.7.2 Stufen

Eine Zustandsmaschine durchläuft folgende Stufen:

1. Die Maschine beginnt im Initialzustand.
2. Die Maschine wartet auf ein Ereignis.
3. Ein Ereignis richtet sich an die Maschine.
4. Wenn das Ereignis in dem aktuellen Zustand nicht akzeptiert wird, wird es ignoriert.
5. Wenn das Ereignis akzeptiert wird, löst es einen bestimmten Übergang aus, der das weitere Verhalten spezifiziert. Der nächste Zustand wird zum aktuellen Zustand und die dazugehörige Aktion wird ausgeführt.
6. Im Folgenden wird Schritt 2 bis 5 so lang wiederholt, bis ein finaler Zustand erreicht wird.

Nähere Informationen zu deterministischen finiten Zustandsmaschinen bieten Mealy [Mea55], Moore [Moo56], Ashby [Ash56], Hopcroft [HU79], Roberts [Rob76] oder Salomaa [Sal85]. Einen guten Überblick über die Theorie der Zustandsmaschine verschafft auch [Min67].

2.7.3 Varianten

Es existieren zwei Varianten von Zustandsmaschinen. Nach George H. Mealy [Mea55] ist ein Zustand ein passives Objekt. Das aktive Objekt ist der Übergang, der eine Aktion zur Folge hat. Jedes akzeptierte Ereignis beeinflusst den aktuellen Zustand der Maschine und bewirkt die Ausführung einer bestimmten Aktion. Mealy-Maschinen werden von den meisten Software-Entwicklungen präferiert [Har88].

Edward P. Moore entwickelte zur gleichen Zeit ein unterschiedliches Modell. Hier stellt der Zustand, dem jeweils eine Aktion zugeteilt ist, den aktiven Bestandteil des Modells dar. Er wird durch eine bestimmte Eingabe gesetzt. Ereignisse verändern den Zustand nicht und so wird er so lang gehalten, bis eine weitere Eingabe den Zustand verändert.

Beide Modelle sind mathematisch äquivalent und können somit in das jeweils andere Modell überführt werden.

2.7.4 Zustandsübergangsdiagramm

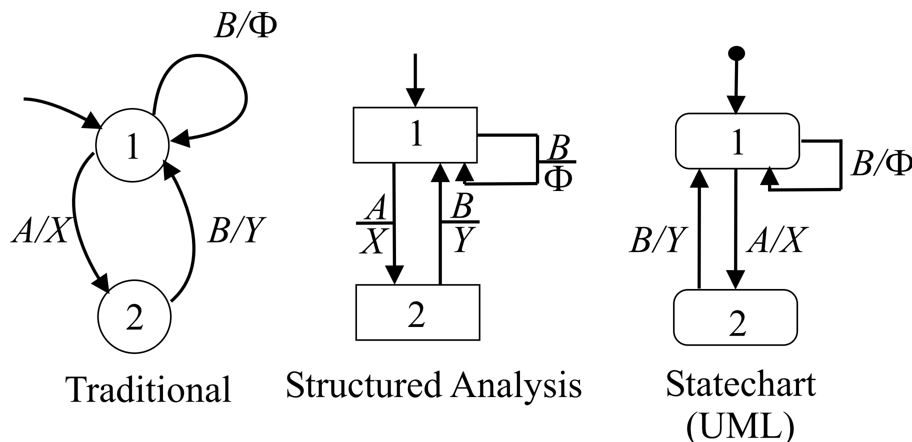


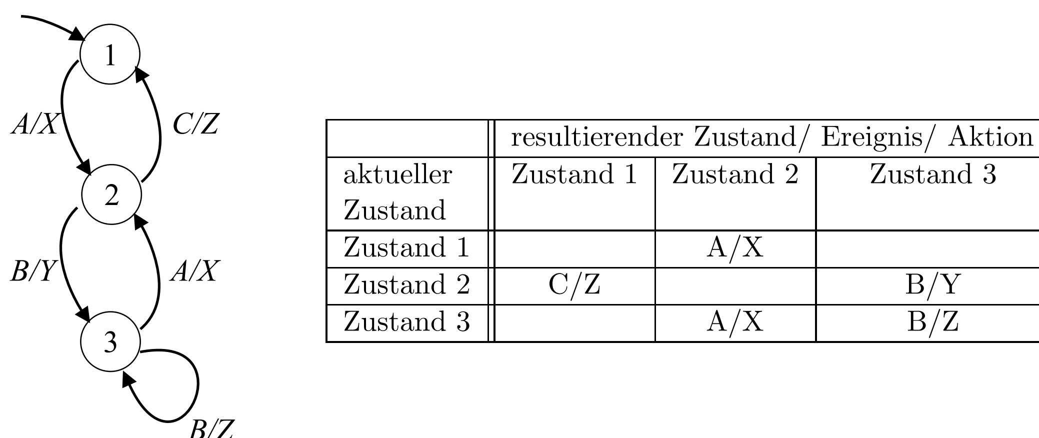
Abbildung 2.3: Die unterschiedlichen Diagramm-Notationen

Zur graphischen Repräsentation von Zustandsmaschinen haben sich im Laufe der Zeit vor allem drei Notationen von so genannten Zustandsübergangsdiagrammen „state transition diagrams“ durchgesetzt [Bin99, S. 180]: *Traditional*, *Statechart* und die *Structured Analysis* (vgl. Abb. 2.3). Zustände werden als Knoten gekennzeichnet, Übergänge als Pfeile zwischen den Knoten und Ereignisse zusammen mit den Aktionen werden seitlich an die Pfeile geschrieben. Ein einzelner Pfeil oder ein Punkt mit einem Pfeil in Richtung Knoten

signalisiert, dass es sich bei diesem Knoten um einen initialen Zustand handelt. Ein Pfeil zu einem Punkt in einem Kreis definiert einen finalen Zustand. Die Zustände werden durchnummeriert, Ereignisse werden mit $A, B, C...$ bezeichnet und Aktionen mit $...X, Y, Z$. Φ definiert eine Null-Aktion. Ein Übergang mit einer Null-Aktion hat keine Aktion zur Folge.

Die traditionelle Notation kennzeichnet einen Zustand durch einen Kreis und enthält gebogene Pfeile. Durch die Structured Analysis Notation entsteht eine graphische Übersicht, die einem Schaltplan ähnelt. Zustände werden durch Rechtecke gekennzeichnet und Pfeile werden rechtwinklig organisiert. Angaben zu einem Übergang werden mittig an einen Strich angeordnet. Die Statechart-Notation symbolisiert Zustände durch Kästchen (abgerundete Ecken). Ereignis und Aktion werden durch einen Schrägstrich an einen Pfeil gezeichnet.

In der Statechart-Notation können Übergänge und Zustände durch Zusammenfassung und die Bildung eines Überzustands (superstate) [Bin99, S. 194] vereinfacht werden. Hierdurch können auch Hierarchien im Diagramm abgebildet werden. Die traditionelle Notation wird in wissenschaftlichen Arbeiten am häufigsten genutzt.



	resultierender Zustand/ Ereignis/ Aktion		
aktueller Zustand	Zustand 1	Zustand 2	Zustand 3
Zustand 1		A/X	
Zustand 2	C/Z		B/Y
Zustand 3		A/X	B/Z

Abbildung 2.4: *links*: Zustandsübergangsdiagramm (traditionell); *rechts*: zugehörige state-to-state-Tabelle

Mit zunehmender Komplexität verringert sich die Übersichtlichkeit des Diagramms und daher ist es ratsam, eine weitere Methode der Darstellung zu wählen. Binder beschreibt in [Bin99, S. 189] den Aufbau einer *state-to-state*-Tabelle (vgl. Abb. 2.4), die akzeptierende Zustände den resultierenden gegenüberstellt. Einen detaillierteren Überblick verschafft eine Erweiterung der Tabelle, das so genannte *expanded state-to-state*-Format. Eine weitere Tabelle, die *event-to-state*-Tabelle, stellt Ereignisse als Reihe und Zustände als Zeile auf [Bin99, S. 189].

2.7.5 Modellierung und Fehler

Das Modellieren einer Zustandsmaschine muss stets gut durchdacht sein. Besonders, wenn komplexere Abläufe abgebildet werden, sollte das Resultat der Modellierung getestet wer-

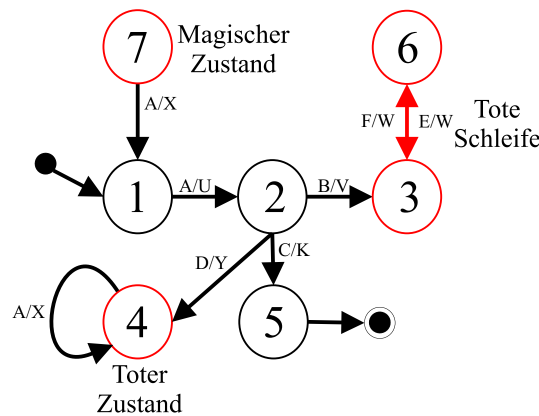


Abbildung 2.5: Fehler in Zustandsmaschinen

den. Hierbei können weniger inhaltliche als syntaktische Fehler detektiert werden.

Ein fehlerhafter Zustand wäre ein so genannter *toter Zustand* (dead state). Wenn sich die Maschine in einem solchen Zustand befindet, führt kein Übergang mehr aus diesem heraus. Ähnliches gilt für eine *tote Schleife* (dead loop) (vgl. Abb. 2.5).

Diese Zustandsarten können in manchen Fällen auch gewollt sein, sind jedoch in der Regel ein Resultat fehlerhaften Designs. Ein weiterer fehlerhafter Zustand ist der *magische Zustand* (magic state) [Bin99, S.183]. Dieser kann von anderen Zuständen aus nicht erreicht werden, besitzt aber einen Übergang zu einem bzw. mehreren Zuständen.

Kombinationen dieser Fehler verschlechtern das Modell zunehmend. Als Beispiel gilt hier ein Zustand, der zwar einen Übergang hat, der aber nur in sich selbst führt. Von außen ist er nicht erreichbar. Wenn man davon ausgeht, dass das Modell nicht nur aus diesem Zustand besteht, kann dieser Zustand nicht gewollt sein, da er auch keine Aufgabe übernimmt. Besteht das Modell nur aus diesem Zustand, so ist die Zustandsmaschine ein eher simples Modell eines noch einfacheren Systems.

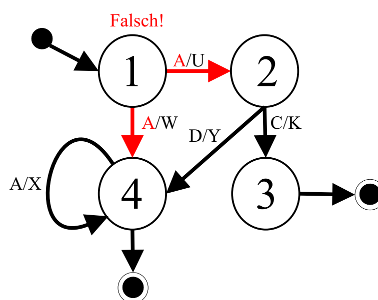


Abbildung 2.6: Fehler in einer deterministischen Zustandsmaschine

Nicht nur die Modellierung der Zustände kann Fehler mit sich bringen, sondern auch die Wahl der Übergänge. In einer deterministischen Zustandsmaschine existieren in einem

Zustand nicht mehrere Übergänge, die vom gleichen Ereignis ausgelöst werden (vgl. Abb. 2.6). Somit ist gewährleistet, dass sich die Maschine zu einem Zeitpunkt nur in einem Zustand befinden kann.

2.7.6 Wächter

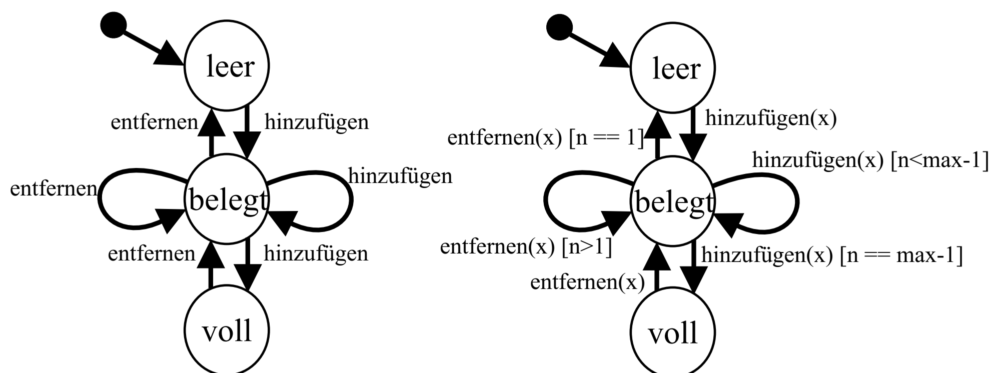


Abbildung 2.7: Zustandsmaschinen-Modell eines Kellerspeichers; *links*: nichtdeterministischer Ansatz; *rechts*: deterministischer Ansatz mit Wächtern

In einem deterministischen finiten Automaten darf, wie soeben beschrieben, kein Zustand existieren, der mehrere Übergänge mit gleichem Ereignis besitzt. Die gestellten Ansprüche an eine Zustandsmaschine können jedoch gerade dies erfordern.

Um diese Situation besser zu verdeutlichen, betrachten wir hier ein Beispiel (vgl. Abb. 2.7): Ein *Kellerspeicher* (stack) mit einer maximalen Anzahl N an Elementen soll durch eine Zustandsmaschine realisiert werden. Im initialen Zustand ist der Speicher leer, die Maschine startet also im Zustand „Leer“. Wird ein Ereignis „Element hinzufügen“ empfangen, so geht die Maschine in einen Zustand „Belegt“ über. Empfängt die Maschine weitere Ereignisse „Element hinzufügen“, so soll sie so lang im Zustand „Belegt“ verweilen, bis sie $N-1$ Elemente beinhaltet. In diesem Fall soll sie in den Zustand „Voll“ übergehen, das Element hinzufügen und dann keine weiteren Elemente mehr aufnehmen. Ein ähnliches Verhalten soll die Maschine bei der Abgabe von Elementen aufweisen. Hier soll sie solange im Zustand „Belegt“ verweilen, bis sie genau ein Element enthält. Beim nächsten Ereignis „Element entfernen“ soll sie in den Zustand „Leer“ übergehen.

Solche Übergänge nach Bedingung werden als *Wächter* (guards) bezeichnet und in die grafische Übersicht als näher definiertes Ereignis gezeichnet. Die Unified Modeling Language (UML^{TM}) Notation [Gro03], die die Statechart-Notation angenommen hat, bezeichnet Wächter folgendermaßen:

Ereignisname *Argumentenliste* [*Wächterbedingung*] / *Aktionsausdruck*

In unserem obigen Beispiel:

Ereignis_Hinzufügen	Element	[n<N-1]	/ return Zustand_Belegt
Ereignis_Hinzufügen	Element	[n==N-1]	/ return Zustand_Voll

Die Bedingung in Worten: Wenn ein Ereignis mit dem Inhalt „Element hinzufügen“ empfangen wird, dann prüfe nach, ob die Anzahl gespeicherter Elemente kleiner als $N-1$ ist. Wenn ja, dann bleibe in dem aktuellen Zustand, wenn nein, dann gehe in den Zustand „Voll“ über.

Andere Notationen bedienen sich eines eigenen Knotens, um einen Wächter zu kennzeichnen. So definiert Page-Jones [PJ95] einen Diamanten und Harel [HG97] einen Punkt als Symbol für einen Wächter.

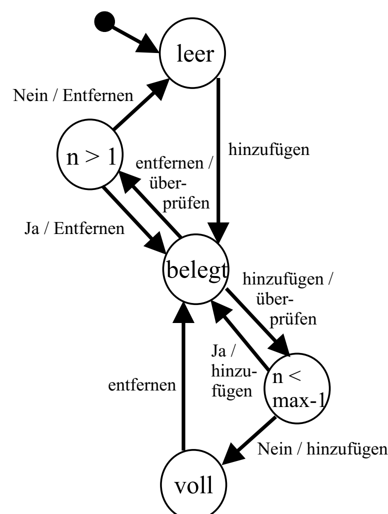


Abbildung 2.8: Realisierung von Wächtern mittels Zwischenzuständen

Eine Realisation dieser Wächter behilft sich eines einfachen Tricks (vgl. Abb. 2.8): Hierbei wird ein Zwischenzustand eingefügt, der als Weiche dient. Die Aktion, die beim Übergang in diesen Zwischenzustand ausgeführt wird, überprüft die Bedingung und sendet das Ergebnis in einem neuen Ereignis an die Zustandsmaschine. Diese überführt nun entsprechend des Ergebnisses den aktuellen Zustand vom Zwischenzustand in den korrekten nächsten Zustand.

Eine Anwendung einer solchen Zustandsmaschine (stack) wäre das Verhalten eines grafischen Winkelvermessungs-Werkzeugs (vgl. Abb. 2.9). Hierbei dürften höchstens $N = 3$ Punkte existieren, die sich zu zwei Geraden formieren. Nachdem der dritte Punkt hinzugefügt wurde, wird der Winkel zwischen den Geraden berechnet und ausgegeben.

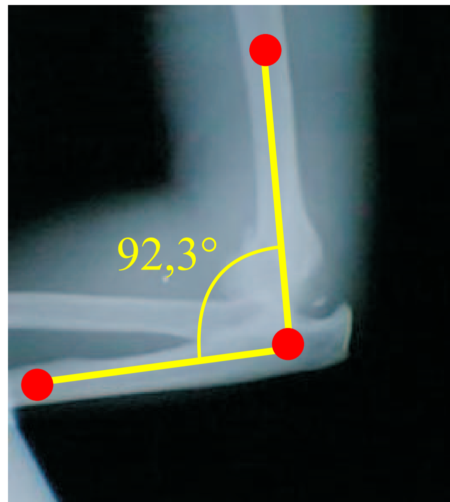


Abbildung 2.9: Beispiel einer Winkelvermessung

2.7.7 Grundregeln für die Erstellung

Für die Erstellung von Zustandsmaschinen gelten folgende Grundregeln [Bin99]:

- Es muss genau ein initialer Zustand definiert sein.
- Es ist mindestens ein finaler Zustand definiert.
- Jeder Zustand ist vom initialen Zustand aus erreichbar.
- Ein finaler Zustand ist von jedem anderen Zustand aus erreichbar.
- Jedes definierte Ereignis muss mindestens einmal von einem Übergang aufgerufen werden.
- Ausgenommen Initial- und Finalzustand besitzt jeder Zustand mindestens einen ankommenden und einen ausgehenden Übergang.
- In einem Zustand sind alle Ereignisse eindeutig, d.h. in einem Zustand existieren nicht mehrere Übergänge, die infolge eines gleichen Ereignisses in unterschiedliche Zustände übergehen (Determinismus).
- Die Zustandsmaschine ist vollständig definiert. Zu jedem Übergang ist ein Ereignis, ein akzeptierender und ein resultierender Zustand angegeben. Zudem ist jedem Übergang eine Aktion zugeordnet (gegebenenfalls auch Null-Aktion Φ).

2.7.8 Vor- und Nachteile

Der Hauptvorteil der Verwendung von Zustandsmaschinen besteht darin, dass ein komplexes System in kleinere Subsysteme zerlegt werden kann und so übersichtlicher wird. Ebenso können Hierarchien durch Zustandsmaschinen modelliert werden.

Als Beispiel betrachten wir ein umfangreiches Bildverarbeitungsprogramm. Wenn wir uns ein solches Programm in Hinblick auf das Verhalten genauer anschauen, so können wir Subsysteme ausfindig machen, die von anderen Subsystemen unabhängig sind. So ist etwa das Verhalten eines Zeichenwerkzeugs unabhängig von dem Verhalten eines Translationswerkzeugs. Bei den meisten Bildverarbeitungsprogrammen besteht nicht einmal die Möglichkeit, beide Werkzeuge parallel zu verwenden, d.h. das Objekt zu verschieben, während es sich noch im Aufbau befindet. Schauen wir näher auf das Zeichenwerkzeug, so können wir weitere Subsysteme erkennen. Das Zeichenwerkzeug verwendet Punkte, so genannte Stützstellen, um einen Kurvenverlauf zu definieren [Far02]. Einen solchen Punkt können wir auswählen. Nun erscheinen an den Seiten, links und rechts, zwei Linien (vgl. Abb. 2.10). Verändern wir die Richtung oder die Länge dieser Linien, so ändert sich auch der Verlauf der Kurve. Diese Linien, auch als Tangenten zu sehen, haben ein eigenständiges Verhalten und können gegebenenfalls auch Eigenschaften des Supersystems „Kurvenverlauf“ verändern.

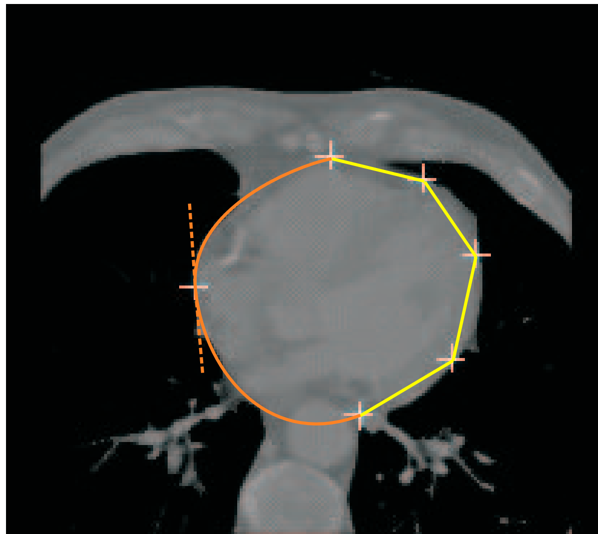


Abbildung 2.10: Einsatz eines Geraden- und eines Bezier-Werkzeugs; Tangente der Bezier-Kurve ist rot-gestrichelt dargestellt

Ein weiterer Vorteil des Einsatzes von Zustandsmaschinen ist die Möglichkeit der Wiederverwendung. Im Beispiel betrachtet könnten wir nun das Objekt „Tangente“ für einen Kurvenverlauf und für eine Winkelvermessung gebrauchen. Das Verhalten ist hier gleich, nur die resultierenden Informationen werden unterschiedlich verarbeitet.

Die Mächtigkeit einer Zustandsmaschine kann aber auch zum Verhängnis werden, wenn

die gestellten Ansprüche an eine Software-Entwicklung kein komplexeres Verhaltensmuster erfordern. In einem solchen Fall sollten eher Bedingungsanweisungen (switch/case; if..then..) verwendet werden, besonders wenn abzusehen ist, welche Kombination von Ereignissen zur Laufzeit entstehen.

2.7.9 Testen

Eine Zustandsmaschine kann auf syntaktische Korrektheit geprüft werden. Dabei entscheiden die Art der Maschine und das Design über die Möglichkeiten, die dem Testverfahren zur Verfügung stehen. Deshalb sollte die Notwendigkeit eines Tests schon in der Designphase mit eingeplant werden. Ansätze zu Testverfahren und auch Checklisten zur Vollständigkeit von Zustandsmaschinen bietet Robert V. Binder in [Bin99].

2.7.10 Software-Entwicklungen mit Zustandsmaschinen

Erich Gamma et. al. beschreiben in [GHJV96, S. 398-409] ein *objektbasiertes Zustandsmuster*. Ein Zustandsmuster ist hier als Implementierungsschema einer Zustandsmaschine zu sehen. Hierbei handelt es sich um ein Objekt, welches in der Lage ist, intern unterschiedliche Zustände zu durchlaufen. In verschiedenen Zuständen verhält sich das Objekt auch unterschiedlich, so dass es von außen den Anschein hat, es würde sich um eine andere Klasse als zuvor handeln.

Ein Zustandsmuster ist gerade dann von Vorteil, wenn das Verhalten eines Objekts zur Laufzeit von vorherigen Eingaben, also vom aktuellen Zustand, abhängt. Eine Funktion, die große mehrteilige Bedingungsanweisungen enthält, spricht ebenfalls für das Verwenden eines Zustandsmusters. Dieses verlagert die Bedingungen in separate Klassen, so dass jeder Objektzustand als ein eigenständiges Objekt behandelt werden kann. Letzteres dient natürlich auch der Übersichtlichkeit und Wiederverwendung von Quellcode. Es beinhaltet aber auch, dass sich eine Vielzahl von Unterklassen bilden.

Tom Cargil verwendet eine *tabellenbasierte Zustandsmaschine* [Car92]. Diese sieht in jedem Zustand eine Tabelle vor, in der alle akzeptierten Ereignisse auf einen Nachfolgezustand abgebildet werden. Hauptvorteil hierbei ist die Möglichkeit, schneller und unkomplizierter Veränderungen an den Tabellen vorzunehmen. Der Nachteil, der daraus resultiert, ist eine geringere Übersichtlichkeit.

Die Firma Protos Software GmbH bietet ein Entwicklungstool namens Trice [Pro03b] an, mit dessen Hilfe ein Softwaresystem grafisch moduliert werden kann. Dabei werden alle aktiven Objekte des Systems in Form so genannter *Aktoren* modelliert. Diese können auch aus weiteren Aktoren zusammengesetzt sein, so dass ein komplexes Konstrukt aus kleineren Bestandteilen hergestellt werden kann. Die erstellten Aktoren stellen die statische Struktur des Systems dar. Das dynamische Verhalten wird über Zustandsgraphen definiert. Dabei

muss für jeden Aktor ein solcher Zustandsgraph erstellt werden, der die eigenen Abläufe und Reaktionen auf Nachrichten von außen beschreibt.

Weitere Beispiele von Software-Entwicklungen mit tabellenbasierten Zustandsmaschinen beschreibt Douglass in [Dou98].

2.7.11 Der Einsatz von Zustandsmaschinen

Zustandsmaschinen können in vielen Bereichen eingesetzt werden. Durch die variable Definierbarkeit des Systems können sie sogar für die Simulation biologischer Regelkreise eingesetzt werden. Häufig finden Mealy-Maschinen Verwendung in interaktiven Zeichenprogrammen. Hier wird ihre Eigenschaft für die Realisation unterschiedlicher Zustände von Zeichenwerkzeugen genutzt. Beispiele für den Einsatz von Zustandsmustern in einem Zeicheneditor-Framework sind HotDraw [Joh92] und Unidraw [VL90].

Die 3D-Grafikbibliothek OpenGL® verwendet eine Zustandsmaschine des Typs Moore, um Eigenschaften global zu setzen. Durch Aufruf von Funktionen kann der Zustand manipuliert werden. Ein Funktionsaufruf `glColor3f(1.0, 1.0, 1.0)` setzt die Eigenschaft „Farbe“ auf weiß. Alle folgenden *Primitive* werden in weiß gezeichnet. Die Farbe bleibt so lang aktuell, bis ein erneuter Aufruf von `glColor` getätigt wird.

Das wohl am häufigsten zitierte Beispiel einer Zustandsmaschine ist ein TCP-Verbindungsprotokoll [JJ91] [GHJV96].

2.8 Undo/Redo

In 2.6 haben wir bereits die komplexe Interaktion, die wir in medizinischen Applikationen vorfinden, besprochen. Bei einer solch komplizierten Interaktion müssen wir davon ausgehen, dass der Benutzer Fehler macht. Um diese Problematik in unserem Softwaredesign mit zu berücksichtigen, müssen wir uns ein Konzept überlegen, wie ein Befehl, der vom Benutzer eingegeben wurde, wieder rückgängig gemacht werden kann.

In diesem Kapitel treten Begriffe wie *Undo* und *Redo* sehr häufig auf. Dies ist aufgrund des Themas unvermeidbar. Andere zentrale Begriffe wie *Operation* und *Befehl*, die im ersten Moment gleich erscheinen, hier jedoch Unterschiedliches bedeuten, werden im Glossar differenziert.

Das folgende Unterkapitel beschäftigt sich mit den Prinzipien des Undos. Zu Beginn wird die Entstehung von Fehlern betrachtet und der mögliche Umgang mit diesen geschildert. Ein Rückblick auf die Entstehung des Undos folgt im Anschluss. Hiernach werden gängige Undo-Modelle beschrieben und Vorschläge zum Nachlesen weiterer Modelle genannt. Zum Schluss wird geklärt, welche Anforderungen erfüllt sein müssen und was bei der Realisierung

beachtet werden sollte, um die Funktionalität eines Undos in einer Software-Entwicklung zu nutzen.

2.8.1 Benutzerfehler und ihre Revidierungsmöglichkeiten

Fehler können während des gesamten Programmablaufs geschehen. Um die Benutzerfreundlichkeit in so einem Fall zu wahren, sollte es dem Benutzer möglich sein, zu jedem Zeitpunkt in den Programmablauf einzugreifen. Der Programmablauf kann hier in einzelne Berechnungen unterteilt werden, von denen einige rechenintensiv sein können (Filter etc.).

Ein Fehler kann zum einen vor Ausführung einer Berechnung vorkommen, wie etwa bei falscher Eingabe von Parametern. Hier ist eine *Flucht-Funktion* (Escape) von Vorteil, bei der die Berechnung noch vor Beginn abgebrochen wird. Zum zweiten kann ein Fehler während der Ausführung einer Berechnung entstehen. Mit einer *Abbruch-Funktion* (Cancel) kann der Benutzer die Berechnung im Betrieb abbrechen und muss so nicht auf die Beendung der Berechnung warten. Wird ein Fehler aber erst nach Vollendung der Berechnung entdeckt, so stellt ein *Undo-Funktion* den Zustand vor der Berechnung wieder her.

Speziell die letzte Funktion steigert die Benutzerfreundlichkeit um ein Vielfaches. Wenn nun vom Benutzer eine komplexe Interaktion für die Berechnung gefordert wird, ist diese Undo-Funktionalität unersetzlich.

2.8.2 Die Geschichte des Undos

Die ersten Wiederherstellungstechniken wurden in Datenbankprogrammen eingesetzt. Die so genannte *Rollback-Funktion* diente dazu, die Konsistenz einer Datenbank wiederherzustellen, nachdem eine Transaktion aufgrund eines Systemfehlers oder Ähnlichem nicht ausgeführt werden konnte. Die Rollback-Funktion brauchte nicht besonders effizient in Bezug auf Ausführungsgeschwindigkeit oder Benutzerfreundlichkeit zu sein, da eine solche Ausnahmesituation selten auftrat [JCS84].

Computer wurden lange Zeit nur im Stapelverarbeitungsbetrieb (Batch) benutzt. Hierbei hatte der Benutzer nach Start des Programms keine Möglichkeit, in den Programmablauf einzugreifen. Deshalb stellte man Techniken ähnlich der Rollback-Funktion her, durch die der Benutzer den Zustand vor einem Befehl speichern konnte. Durch das Laden des vorherigen Zustands nach falscher Berechnung konnte das System wieder in den letzten Zustand überführt werden [Thi90].

Mit dem Wechsel zu den interaktiven Anwendungen bestand die Möglichkeit der interaktiven Korrektur der zuvor getätigten Eingaben. Zu der Zeit bestand die Mehrzahl der Programme aus Texteditoren [Jr86]. Die erste Implementierung einer Wiederherstellungsfunktion in einer umfangreicheren Anwendung erfolgte in Interlisp, einer Entwicklungsumgebung für die Programmiersprache Lisp [Tei78].

Mit den ersten grafischen Benutzeroberflächen wie Windows® 1.0 (1985) von Microsoft® stieg die Anzahl von Programmen mit Undo-Funktionalität. Mit Windows® 3.00a bildete sich 1990 ein systemweit einheitliches Benutzungsmodell, aus dem sich auch der Vorschlag einer Undo-Funktionalität herauskristallisierte. Von dem Zeitpunkt an wurden die meisten Standardanwendungen mit Wiederherstellungsfunktion ausgestattet. Heutzutage ist diese Funktionalität aus keinem komplexeren Programm mehr wegzudenken und gehört somit zum Standard.

2.8.3 Undo-Modelle

Eine Möglichkeit zur Klassifizierung von Undo-Modellen besteht nach Berlage in der Unterscheidung von Modellen mit *stabilen* und *nicht-stabilen Ausführungseigenschaften* (stable execution property) [Ber94].

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

Eine stabile Ausführungseigenschaft besitzt ein Undo-Modell, wenn es eine Redo-Operation nur in dem Zustand ausführt, in dem die zugehörige Operation ausgeführt wurde, und eine Undo-Operation nur in dem Zustand ausführt, in den die zugehörige Operation führte. Undo-Modelle, die zulassen, dass eine Operation an einer beliebigen Stelle der Befehlsgeschichte gespeichert wird, verstoßen hiergegen.

Besitzt ein Undo-Modell eine stabile Ausführungseigenschaft, so hat das erhebliche Vorteile. Der wichtigste davon ist, dass keine Unstimmigkeiten im Ablauf des Programms entstehen können. Dies wäre dann der Fall, wenn eine Undo- oder Redo-Operation in einem Zustand ausgeführt wird, in dem sie nicht definiert ist. Diese Eigenschaft erfüllen mit Ausnahme des *uneingeschränkten linearen Undo-Modells* (vgl. S. 26) alle einfachen und linearen Modelle.

In allen Undo-Modellen werden ausgeführte Befehle in einer *Befehlsgeschichte* (command history) gespeichert. Diese kann entweder aus einer Liste bestehen, in der man mit einem *Befehlszeiger* auf- und abfährt, oder aus einer Undo- und einer Redo-Liste aufgebaut werden. Ein auszuführender Befehl wird in der Undo-Liste gespeichert. Führt der Benutzer ein Undo aus, so wird der entsprechende Undo-Befehl aus der Undo-Liste genommen, ausgeführt und in die Redo-Liste geschrieben. Beide Verfahren werden bei der Realisation einfacher Undo-Modelle benutzt.

Eine besondere Funktionalität besteht darin, diese Befehlsgeschichte auf einem persistenten Speichermedium, z.B. einer Festplatte oder einer Diskette, zu speichern. Indem bei

Programmstart alle Befehle in der Geschichte der Reihe nach ausgeführt werden, kann nach einem Systemabsturz wieder zu dem Zustand vor Absturz zurückgekehrt werden [JCS84].

Um einen Speicherüberlauf zu verhindern, muss die Befehlsgeschichte auf ihre Größe begrenzt werden. Die maximale Größe kann sich nach der Art des Undo-Modells richten sowie von der Realisierung des Modells abhängen. Setzt sich eine Operation nur aus den nötigsten Elementen zusammen, so verringert das den Speicherbedarf pro Operation.

Selbstverständlich richtet sich die maximale Größe der Befehlsgeschichte auch nach dem zur Verfügung stehenden Speicherplatz. Entweder muss dieser zum Programmstart überprüft oder durch eine Benutzerabfrage berechnet werden. Ein übliches Verfahren ist es, den Benutzer die Anzahl der möglichen Undos definieren zu lassen. Multipliziert mit der Speichergröße pro Operation ergibt das dann den zu reservierenden Speicherplatz. Wird der Speicherplatz während der Ausführung des Programms überschritten, so kann je nach Undo-Modell die Befehlsgeschichte teilweise gelöscht werden. Bei einem *eingeschränkten linearen Undo-Modell* (vgl. S. 25) wird die letzte Operation gelöscht (vgl. Abb. 2.11). Bei einem *linearen Modell mit Befehlsbaum* (vgl. S. 27) kann entweder gleich verfahren oder durch eine Benutzerabfrage ein zu löschender Ast bestimmt werden (vgl. Abb. 2.12).

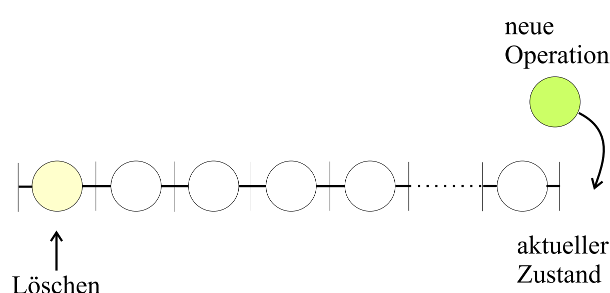


Abbildung 2.11: Lineares Löschverfahren bei Speicherüberlauf; jeder Kreis symbolisiert ein Operationsobjekt

Im Folgenden wird als Beispiel für ein einfaches Modell auf das *Single-Undo-Modell* eingegangen. Komplexere, lineare Modelle werden anschließend beschrieben. Darauf folgt eine grobe Beschreibung weiterer Undo-Modelle mit Hinweisen zum Nachlesen. Zum Schluss des Kapitels legen wir das Augenmerk auf die Anforderungen, die für eine Undo-Funktionalität erfüllt werden müssen, und diskutieren, unter welchen Umständen ein bestimmtes Modell gewählt werden sollte.

Single-Undo-Modell

Das *Single-Undo-Modell* zählt zu den einfachen Undo-Modellen, denn es wird nur der letzte Befehl in der Befehlsgeschichte gespeichert. Dadurch hat der Benutzer nur die Möglichkeit, die letzte Operation rückgängig zu machen, und diese gegebenenfalls auch zu wiederholen. Zwar hat dieses Modell für den Benutzer nur sehr bedingt einen Nutzen, jedoch ist es

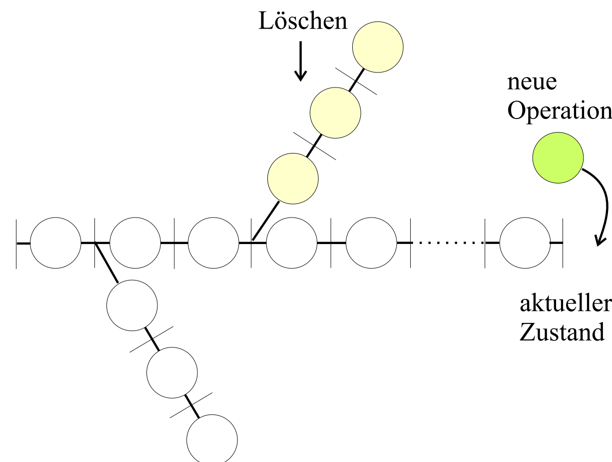


Abbildung 2.12: Baumbasiertes Löschverfahren bei Speicherüberlauf; eine Regel oder der Benutzer entscheidet über den zu löschenden Ast

dafür am besten geeignet, ein Programm nachträglich mit einer Undo-Funktionalität auszustatten. Die Anpassungsmaßnahmen halten sich in einem kleinen Rahmen. Da es hier nur nötig ist, den letzten Befehl rückgängig zu machen, kann es schon ausreichen, vor jeder Ausführung eines Befehls eine Kopie vom Speicherinhalt des Programms herzustellen. Ruft der Benutzer Undo auf, so kann der aktuelle Speicherinhalt mit der Kopie überschrieben werden. Somit haben wir den vorherigen Zustand des Programms wiederhergestellt. Ist abzusehen, dass der Speicherinhalt des Programms zu groß für das Anlegen einer Kopie sein wird, so muss ein anderes Verfahren zur Wiederherstellung des letzten Zustands genutzt werden.

Eingeschränktes lineares Undo-Modell

	ausgeführte Befehle	Befehlsgeschichte	wirksame Befehle	Bemerkung
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 löscht Redo von c

Tabelle 2.1: Beispiel zum Ablauf eines eingeschränkten linearen Undo-Modells [Ost03]; a-e: Befehle, U: Undo, R: Redo, |: Position des Befehlszeigers

Das *eingeschränkte lineare Undo-Modell* (restricted linear Undo) ist leicht zu bedienen, dem Benutzer transparent und wird deshalb auch in den meisten Anwendungen angeboten. Im Gegensatz zum *Single-Undo-Modell* können mehrere Operationen in einer Befehlsgeschichte gespeichert werden. Die Meta-Befehle, Undo und Redo, traversieren hier der Reihe

nach durch die Befehlsgeschichte [Ber94]. Wird eine Kette von Operationen ausgeführt, ein Teil davon wieder rückgängig gemacht und im Anschluss eine neue Operation ausgeführt, so werden die für ein Redo nötigen Informationen gelöscht (vgl. Tab. 2.1). Das Modell bietet dem Benutzer ausreichend Möglichkeit, getätigte Fehler rückgängig zu machen.

Uneingeschränktes lineares Undo-Modell

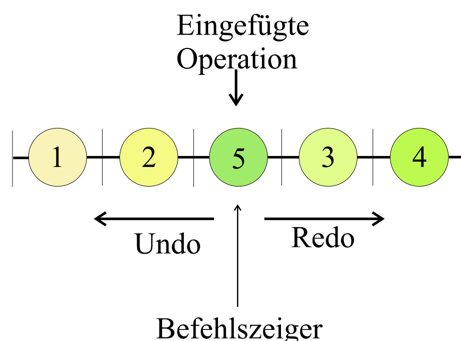


Abbildung 2.13: Uneingeschränktes lineares Undo-Modell; die Reihenfolge der Operationen ist farblich gekennzeichnet und nummeriert

Das *uneingeschränkte lineare Undo-Modell* wird nach Vitter [Vit84] nur lineares Undo-Modell genannt. Zur Differenzierung erweitert Ostermann [Ost03] den Begriff um das Adjektiv „uneingeschränkt“. Im Unterschied zum *eingeschränkten linearen Modell* wird hier die Redo-Liste nach einem neuen Befehl nicht gelöscht. Es erlaubt vielmehr, dass eine neue Operation in die Befehlsgeschichte an der Position des Befehlszeigers eingefügt wird. Ein Meta-Befehl Redo, der nach einem neuen Befehl noch immer angeboten wird, führt die an anderer Stelle zurückgenommenen Operationen erneut aus (vgl. Abb. 2.13). Dies verstößt gegen die stabile Ausführungseigenschaft (vgl. S. 23) und so muss vor einem Redo die Gültigkeit des Meta-Befehls oder der Operation überprüft werden.

	ausgeführte Befehle	Befehlsgeschichte	wirksame Befehle	Bemerkung
1.	abc	abc|	abc	
2.	abcUU	a|bc	a	2x Undo hebt c & b auf
3.	abcUUd	ad|	ad	d wird eingefügt
4.	abcUUdRR	adbc|	adbc	2x Redo stellt b & c nach d wieder her

Tabelle 2.2: Beispiel zum Ablauf eines uneingeschränkten linearen Undo-Modells

Allerdings erlaubt das Modell nicht, eine beliebige Operation in der Befehlsgeschichte mit dem Meta-Befehl Undo rückgängig zu machen. Diese Funktionalität bieten nur *nichtlineare*

oder *selektive Modelle* (näheres hierzu auf S. 28). Wie man am Ablauf, dargestellt in Tabelle 2.2, erkennen kann, ist das Verhalten des Modells eher gewöhnungsbedürftig und kann bei einem unerfahrenen Benutzer zu Verwirrung führen.

Lineares Undo-Modell mit Befehlsbaum

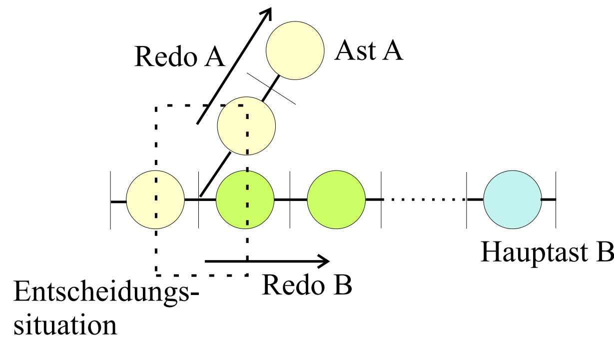


Abbildung 2.14: Lineares Undo-Modell mit Befehlsbaum

Das *lineare Undo-Modell mit Befehlsbaum* (history tree) [Ber94] baut auf dem Verfahren des (uneingeschränkten) linearen Undo-Modells auf. Jedoch wird hier ein Ansatz gewählt, der nicht gegen die stabile Ausführungseigenschaft verstößt (vgl. S. 23).

Um alle Redo-Informationen beizubehalten, müssen wir die Operationen nach ihrer Reihenfolge und Zugehörigkeit sortieren. Dies wird durch eine Baumstruktur realisiert, in der ein neuer Ast für eine neue Operation angelegt wird, nachdem ein Undo-Befehl ausgeführt wurde. Abbildung 2.14 verdeutlicht die Struktur und Tabelle 2.3 erläutert den Ablauf.

	ausgeführte Befehle	Befehlsgeschichte	wirksame Befehle	Bemerkung
1.	abc	abc|	abc	
2.	abcUU	a|bc	a	2x Undo hebt c & b auf
3.	abcUUd	abc \d|	ad	d wird in einem neuen Zweig angelegt
4.	abcUUdUR*	ab?c \d?	ab $\vee$ ad	Undo hebt d auf, Redo muss näher spezifiziert werden

Tabelle 2.3: Beispiel zum Ablauf eines linearen Undo-Modells mit Befehlsbaum; * symbolisiert eine Entscheidungssituation

Durch die Baumstruktur entstehen Doppeldeutigkeiten bei dem Meta-Befehl Redo. Befindet sich der Befehlszeiger in einer Position, an der zwei Befehlsäste zusammenlaufen, ist nicht klar, mit welchem Ast der Benutzer fortsetzen will. Klarheit kann eine entsprechende Abfrage schaffen, in der der Benutzer einen Ast wählt. Ob die beschriebene Besonderheit

einen Vorteil oder einen Nachteil für die Funktionalität darstellt, ist in Zusammenhang mit den Anforderungen an das Undo-Modell zu klären.

Weitere Undo-Modelle

Prakash et al. beschreiben das *History-Undo-Modell* [PK94], welches als Erweiterung des schon erläuterten *Single-Undo-Modells* gilt. Hier werden mehr als nur eine Undo-Operation in der Befehlsgeschichte gespeichert. Die Funktionsweise ist allerdings von der des *eingeschränkten linearen Modells* unterschiedlich und laut Ostermann sehr gewöhnungsbedürftig [Ost03]. Zwei der bekanntesten Programme mit History-Undo-Modell sind die Unix-Texteditoren „vi“ und „(X)Emacs“ [Pro03a]. Zusätzliche Informationen zum History-Undo-Modell sind bei Prakash et al. [PK94] und bei Ostermann [Ost03] zu finden.

Ein weiteres Modell stellen Archer et al. in [JCS84] mit *Skript-Modellen* vor. Eines der Modelle, das *Truncate/Reappend-Modell*, sieht vor, dass mit Hilfe von Meta-Befehlen die Befehlsgeschichte umgestellt werden kann. Dieses Modell besitzt nach Berlage keine stabile Ausführungseigenschaft [Ber94, S. 274].

Vitter beschreibt in [Vit84] das so genannte *US&R-Modell*, bei dem es zum einen möglich ist, eine Operation in die Befehlsgeschichte einzufügen, zum anderen bei einem neuen Befehl die Redo-Befehlsgeschichte nicht gelöscht wird. Mit den drei Befehlen *Undo*, *Skip* und *Redo* kann der Benutzer sich in der Befehlsgeschichte bewegen. Das US&R-Modell besitzt ebenso keine stabile Ausführungseigenschaft.

Weitere Modelle werden von Yang (*Triadic-Modell*) [Yan88] und Berlage (*direktes selektives Undo-Modell*) [Ber94] vorgestellt. Eine gute Übersicht über Undo-Modelle bietet Ostermann in [Ost03].

2.8.4 Anforderungen

Eine Anforderung einer Undo-Realisation bestimmt, dass das Verhalten unseres Undo-Modells transparent für den Benutzer sein muss. Wenn die angebotene Wiederherstellung mehr verwirrend als unterstützend wirkt, trübt dies das Vertrauen des Benutzers. Somit sinkt die Benutzerfreundlichkeit (usability) des Programms erheblich.

Wenn eine Operation nicht zurückgenommen werden kann oder auf anderem Wege Undo-/Redo-Information gelöscht werden soll, sollte der Benutzer darüber vor Ausführung der Operation in Kenntnis gesetzt werden. Gleichzeitig kann dem Benutzer angeboten werden, das Projekt oder die Umgebung zu speichern.

Ferner sollte überprüft werden, welche Operationen in die Befehlsgeschichte aufgenommen werden. Lese-Operationen, die keine Veränderungen vornehmen, kommen hierfür nicht infrage [Yan88]. Operationen, die außerhalb des Einflusses wirken (z.B. E-Mail verschicken), können auch nicht zurückgenommen werden, sollten also auch nicht gespeichert werden. Raskin legt fest, dass Benutzereingaben wie das so genannte *Scrollen* der Maus oder das

Selektieren von beliebigen Elementen nicht in die Befehlsgeschichte gehören [Ras00]. Myes et al. jedoch beschreiben den Nutzen der Speicherung darin, dass hiermit auch komplexe Selektionen wiederhergestellt werden können [MK96].

Für die Ausführung einer Operation werden spezielle Informationen benötigt und so können wir eine Operation in folgende Elemente unterteilen:

- Art der Operation
- Parameter, die für die Ausführung der Operation nötig sind
- Zeitstempel, laufende Nummer oder andere Identifikatoren
- sonstige Informationen

Damit wir eine Operation zu einem späteren Zeitpunkt rückgängig machen können, müssen wir alle nötigen Elemente in der Befehlsgeschichte speichern. So überdauern die Elemente jeder Operation ihre Ausführung, was eine wesentliche Anforderung für die Realisierung darstellt.

2.8.5 Die Realisierung von Undo-Modellen

Zwei Möglichkeiten für die Realisation einer Undo-Funktionalität sind die *Strategie des kompletten erneuten Ausführens* und die der *inversen Operation* bzw. des *inversen Befehls*.

Bei der *Strategie des kompletten erneuten Ausführens* (complete return strategy) [JCS84] beginnt die Undo-Prozedur am Start des Programms, lädt evtl. die zu bearbeitende Umgebung und führt alle Operationen der Befehlsgeschichte der Reihe nach aus, bis sie an der letzten Operation angelangt ist. Diese wird ausgelassen und so befindet sich das Programm im vorherigen Zustand. Der Speicher der Befehlsgeschichte darf nicht limitiert sein, denn es werden alle bisher ausgeführten Operationen benötigt und dürfen nicht gelöscht werden. Die Strategie führt zu einer sehr langsamen Undo-Realisation. Wenn jedoch ein Programm nachträglich erweitert werden soll, ist sie relativ leicht umzusetzen.

Varianten dieser Strategie sehen das Abspeichern von Zuständen in gewissen Abständen vor, so dass zum einen die Befehlsgeschichte limitiert sein darf und zum anderen der Prozess des erneuten Ausführens beschleunigt werden kann [JCS84].

Die *Strategie der inversen Operation* (inverse command strategy) [JCS84] sieht für jede Operation eine zugehörige inverse Operation vor. Im Falle eines Undos wird die inverse Operation ausgeführt und im Falle des Redos wird die Operation wiederholt.

Für die Erzeugung einer inversen Operation können entweder die Parameter invertiert oder die Art der Operation herangezogen werden (vgl. Elemente der Operation in 2.8.4 und Tab. 2.4).

Element	Operation	inverse Operation
Art der Operation	Punkt löschen	Punkt hinzufügen
Parameter	um X verschieben	um $-X$ verschieben

Tabelle 2.4: Beispiele für inverse Operationen

Allerdings können die inversen Operationen nicht immer so einfach zugeordnet werden (Doppeldeutigkeit). In dem Beispiel könnte die inverse Operation "neuen Punkt setzen" ebenso die Aufgabe der inversen Operation "Punkt hinzufügen" übernehmen.

Werden die Operationen detailliert genug (feingranulär) definiert, so unterstützt das die Zuordnung. Zum Beispiel wäre die Operation „3-Punkte-Objekt löschen" grob, dagegen die Operationen „Punkt 1 löschen", „Punkt 2 löschen", „Punkt 3 löschen" fein definiert.

Eine Alternative zu der inversen Operation ist der *inverse Befehl*, der durch seine Ausführung eine Kette von Operationen rückgängig machen kann. Ein Beispiel hierzu ist ein inverser Befehl "selektierte Gruppe hinzufügen" zu einem Befehl "selektierte Gruppe löschen". Abermals entsteht hier durch einen Befehl "neue Gruppe anlegen" die Gefahr der Überschneidung. Wenn man diese Überschneidungen nicht unterbindet, muss zumindest noch in der Designphase einer Software-Entwicklung darauf geachtet werden, dass klare Richtlinien für inverse Operationen oder Befehle definiert werden.

Die Menge aller möglichen Operationen und Befehle ist bei einem komplexen Programm zu groß, um davon auszugehen, dass für jede Operation/Befehl eine inverse Operation/-Befehl gefunden werden kann. Ein *Kantendetektionsfilter* wirkt sich auf das gesamte zu verarbeitende Bild aus und hebt Kanten im Bild hervor. Da dadurch Informationen im Bild verloren gehen, die wir nicht wiederherstellen können, können wir einen inversen Filter nicht sicher definieren. Auch wenn die Operation noch so feingranulär gewählt wird und wir jede einzelne Veränderung des Bildes in unserer Befehlsgeschichte speichern, kann das die maximale Größe unserer Befehlsgeschichte übersteigen. In einem solchen Fall kann nicht nur die soeben ausgeführte Filteroperation nicht zurückgenommen werden, sondern muss auch auf alle vorherigen Undo-Befehle verzichtet werden. Wir müssen also auch berücksichtigen, dass es Operationen gibt, die nicht undo-fähig sind.

Gamma et al. beschreiben in „Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software" ein Befehlsmuster, mit dem eine Undo-Funktionalität realisiert werden kann [GHJV96, S. 78ff.]. Durch das Befehlsmuster wird die Anforderung erfüllt, die besagt, dass die Informationen jeder Operation ihre Ausführung überdauern müssen (vgl. 2.8.4). Ferner deklarieren sie in einem Objekt „Befehl" eine Methode „IstUmkehrbar", die einen *boolschen Wert* zurückliefert. Der Wert kann zur Laufzeit berechnet werden und so kann

der Undo-Mechanismus entscheiden, ob der Befehl in die Befehlsgeschichte aufgenommen wird oder nicht.

2.8.6 Die Wahl des Modells

Die Wahl des passenden Undo-Modells für eine Applikation kann schwer fallen. Bedacht werden muss zum einen der Benutzerkreis, da ein unerfahrener Benutzer mit der Bedienung eines komplizierten Undo-Modells überfordert sein könnte. In diesem Fall wäre die Funktionalität eines Undo-Modells eher ein Hindernis. Zum anderen muss abgewogen werden, ob nicht schon ein einfaches Modell wie das Single-Undo-Modell oder das eingeschränkte lineare Undo-Modell ausreicht. Komplexere Modelle ohne stabile Ausführungseigenschaft (vgl. 2.8.3) können zu inkonsistenten Zuständen und unvorhergesehenen Ergebnissen führen. Letzteres schadet erheblich der Benutzerfreundlichkeit (vgl. 2.5).

Soll eine schon entwickelte Applikation mit einer Undo-Funktionalität erweitert werden, so ist der Anpassungsaufwand zu beachten. Handelt es sich um eine eher kleine Applikation, so ist das Single-Undo-Modell den anderen vorzuziehen, da hier der geringste Anpassungsaufwand entsteht. Bei größeren Applikationen, in deren Designphase die Undo-Funktionalität nicht mit einbezogen wurde, kommt es darauf an, wie die Informationen jeder Operation organisiert sind (vgl. 2.8.4). Sind diese bereits in einem Operations-Objekt zusammengefasst, so können die gängigen Undo-Modelle nachimplementiert werden. Ist dies nicht der Fall, so sollte die Erweiterung des Programms eher bei der nächsten Entwicklungsstufe erfolgen.

Kapitel 3

Entwurf

Im Folgenden werden die Ansätze beschrieben, durch die das generische Interaktionskonzept mit Undo-Funktionalität realisiert wurde. Dabei werden mehr die Ideen, die hinter der Realisierung stecken, beschrieben als Quellcode gezeigt. Im Kapitel 4 werden dann vereinzelte Auszüge des Quellcodes vorgestellt und es wird noch deutlicher auf die Realisierung eingegangen.

Zunächst folgt eine Beschreibung des Projektplanungs-Modells und eine Ablaufbeschreibung der Planung. Im Anschluss wird das generische Interaktions-Modell erklärt. Mit der Beschreibung der Undo-Realisation endet dieses Kapitel.

3.1 Planung

Das Projekt wurde nach dem *Spiralmodell von Boem* geplant [Boe86]. Zu Beginn wurden allgemeine MITK-Ziele festgelegt, die in einzelne Gebiete (Datenhaltung, Interaktion etc.) untergliedert wurden. Kleinere Projektgruppen wurden gebildet, die die einzelnen Ziele näher untersuchten. Sie standen im ständigen Informationsaustausch untereinander, um den Blick auf das Gesamte nicht zu verlieren. Die Projektgruppe „Interaktion“ und die Gruppe „Undo“, in denen ich tätig war, einigten sich, die Gebiete zunächst getrennt zu betrachten, um sie dann in einem späteren Zyklus gemeinsam zu modellieren. Die Interaktion hängt mit Undo insofern zusammen, dass die Änderungen der Daten, die durch die Interaktion vorgenommen werden, vom Undo widerrufbar sein müssen. Die Fusion „Interaktion“ mit „Undo“ sollte ein gemeinsames Konzept hervorbringen, welches später mit den übrigen Gruppen in Einklang gebracht werden sollte.

Die Ziele der Interaktion wurden aus den gestellten Anforderungen abgeleitet und es wurden Alternativen sowie Rahmenbedingungen formuliert. Mit den Zielen des Undos wurde gleichermaßen verfahren.

Anhand von Beispielen wurde anschließend die Realisierbarkeit überprüft und die resultierenden Risiken besprochen. Tafelbilder verdeutlichten die Beispiele. Hieraus ergaben sich neue Erkenntnisse, die einen Ansatz komplettierten oder zu neuen Varianten führten.

Nach dem Entschluss für ein Konzept der Interaktion und des Undos wurde die Fusionierbarkeit beider Konzepte überprüft. Kleine Ergänzungen, wie z.B. benötigte Parameter, wurden dem Konzept hinzugefügt, so dass ein schlüssiges Interaktionskonzept mit Undo-Funktionalität entstand. Parallel wurden immerzu Alternativen gebildet und Risiken abgewogen.

Im nächsten Schritt wurden die übrigen Gruppen hinzugezogen, um die Umsetzung im gesamten Projekt zu diskutieren. Das gemeinsame Konzept wurde aus der Menge aller Gruppen-Konzeptionen gebildet. Hierfür mussten abermals kleinere Änderungen vorgenommen werden, die jedoch das Grundkonzept nicht veränderten.

Nach dem Finden eines gemeinsamen Konsens wurde in weiteren Zyklen ein detaillierter Software-Entwurf mit der Software-Entwicklungsplattform Rational Rose von IBM [Qua98] erstellt. In den folgenden Zyklen wurden das Vorgehen während der Implementierung besprochen, Konventionen festgelegt und ein Zeitplan für die Implementierung erstellt.

Um das Konzept zu testen, wurden Applikationen entwickelt, die die implementierten Methoden testeten. Zum Zeitpunkt der Erstellung dieser Arbeit befand sich das Projekt in der Endphase des Testens.

3.2 Entwurf des generischen Interaktionsmodells

In der Planungsphase wurden viele Ansätze besprochen. Zum einen wurde der Einsatz eines Interpreters diskutiert, der das komplexe Verhalten realisieren sollte. Der Ansatz wurde jedoch wieder verworfen, da die Realisation und der spätere Umgang zu komplex erschienen. Zu lange Einarbeitungszeit der neu hinzukommenden Entwickler und die schwer zu realisierende Übersichtlichkeit, die die Umsetzung der komplexen Interaktion forderte, sprachen gegen dieses Modell.

Ein Ansatz, der den Einsatz einer Zustandsmaschine vorsah, wurde von der Planungsgruppe als vorteilhaft angesehen. Da sich das unterschiedliche Verhalten der Interaktion leicht in Zuständen und Übergängen abbilden lässt, schien der Ansatz auch gut realisierbar zu sein. In jedem Zustand gelten Regeln, die mittels Übergängen frei definiert werden können. Eine Zustandsmaschine kann auch zusammengefasst werden und in einer anderen Maschine als Zustand aufgeführt sein. So kann die komplexe Interaktion in kleinere Teil-Interaktionen zerlegt werden. Außerdem kann hierdurch eine Zustandsmaschine mehrfach genutzt werden, da davon ausgegangen werden kann, dass die gesamte Interaktion viele gleiche Teil-Interaktionen beinhalten wird. Ferner können Zustandsmaschinen hierarchisch geordnet werden, so dass auch ihre Verwaltung realisierbar ist.

Ein kurzes Beispiel hierzu: Eine Zustandsmaschine, die an oberster Stelle der Hierarchie steht, kümmert sich um die globalen Ereignisse. In ihrem Verhalten ist definiert, in welchem Zustand sie die Ereignisse an welche Zustandsmaschinen weiterleitet. Diese lokalen

Zustandsmaschinen können dann das Ereignis mit ihrem spezifischen Verhalten verarbeiten und gegebenenfalls Daten (z.B. Punktkoordinaten) ändern. Eine detailliertere Darstellung bietet der Abschnitt 3.2.6.

Für die Art der Umsetzung wurde der Ansatz der *Mealy-Maschine* gewählt, da mit ihm am besten das Schema der Interaktion abgebildet werden kann (vgl. 2.7.3). Bei dem Ansatz hat jedes Ereignis Auswirkung auf den aktuellen Zustand der Maschine.

Der Einsatz der Alternative, der *Moore-Maschine*, wäre möglich, aber umständlich. Hier wirkt sich nicht jedes Ereignis auf den Zustand aus, sondern eine bestimmte Eingabe verändert den Zustand. Eine Aktion ist somit einem Zustand und nicht wie bei der Mealy-Maschine einem Übergang zugeordnet. Für die Modellierung des Verhaltens einer Interaktion ist der Einsatz der Moore-Maschine daher aufwändiger.

Für die Realisation der Zustandsmaschine kamen zwei Varianten infrage: *tabellenbasierter* oder *objektbasierter Ansatz* (vgl. 2.7.10).

Gamma et al. beschreiben den Hauptunterschied zwischen dem objektbasierten und dem tabellenbasierten Ansatz folgendermaßen:

> „Das Zustandsmuster modelliert zustandsspezifisches Verhalten, während der tabellengesteuerte Ansatz sich auf die Definition von Zustandsübergängen konzentriert.“

Quelle: [GHJV96, S. 403]

Die Anforderung an unsere Realisation besagt, dass sie zum einen schnell anzupassen, zum anderen übersichtlich sein soll. Der Vorteil des tabellenbasierten Ansatzes liegt in seiner schnellen Veränderbarkeit aufgrund der Definition der Zustände und ihrer Übergänge in einer Tabelle, der Nachteil aber in seiner Unübersichtlichkeit. Die Suche nach einem fehlerhaften oder nach einem bestimmten Zustand, der verändert werden soll, wird durch einen objektbasierten Ansatz erleichtert. Jedoch ist dieser wiederum nicht so schnell veränderbar.

Beide Ansätze haben gesuchte Vorteile, aber auch folgenschwere Nachteile. So entstand die Frage, ob es nicht möglich sei, die Vorteile beider zu kombinieren und die Nachteile zu eliminieren. Die Antwort brachte folgenden Ansatz: Zur Laufzeit wird eine objektbasierte Zustandsmaschine nach einer in einer Datei (Tabelle o.ä.) definierten Form aufgebaut.

Über die generische Realisation (vgl. 2.4) des Ansatzes kann der Aufbau der Zustandsmaschine vor Laufzeit in der Datei beliebig verändert werden. Der Entwickler jedoch arbeitet mit ihm schon vertrauten Objekten, die die Übersichtlichkeit fördern. So kann ein Objekt „Zustand“ definiert werden, das seine Übergänge beinhaltet. Zum Beispiel kann der Entwickler im Quellcode einen spezifischen Zustand nach seinen Übergängen fragen, um an das akzeptierende Ereignis zu gelangen. Außerdem kann er sich über Zeiger in der Zustandsmaschine fortbewegen und auf unübersichtliche Spalten-/Zeilenangaben verzichten.

Die Realisation der Zustandsmaschine ist von der Definition klar getrennt, d.h. es werden keine festen Abhängigkeiten in der Zustandsmaschine implementiert. Vielmehr wird das Konstrukt einer Zustandsmaschine realisiert, mit der Möglichkeit, das Verhalten der Maschine in einer unabhängigen Definition zu modellieren. Die Definition wird in einer Datei geordnet, so dass die Übersichtlichkeit gewahrt wird. XML [Con03] wird für den Aufbau der Tabelle bzw. der Definition der Zustände und Übergänge verwendet (vgl. 3.2.4).

Bei dem Entwurf wurde die Möglichkeit des Tests (vgl. 2.7.9) berücksichtigt. Methoden, die hierfür benötigt werden, wurden in den Aufbau der Objekte einbezogen. In 4.2.3 wird näher auf die Realisation eingegangen.

Der Ansatz stellte sich als ideal heraus. Er realisierte alle Anforderungen und bestand alle möglichen, durch Interaktions-Beispiele simulierten Tests. Der Ansatz wurde komplettiert und verfeinert. Währenddessen wurden Alternativen gesucht, jedoch kein neuer Ansatz gefunden, so dass der Ansatz des generischen Interaktions-Modell für die Realisierung der Anforderungen gewählt wurde.

Nach diesem Entschluss wurde das Konzept mit dem des Undos kombiniert.

Im Folgenden wird das generische Interaktions-Modell detaillierter beschrieben. Zur besseren Lesbarkeit werde ich Präsens anstelle des Präteritums verwenden. Das Modell ist bereits entworfen, implementiert und wird für die Entwicklung von Applikationen genutzt.

3.2.1 Zustandsmaschine

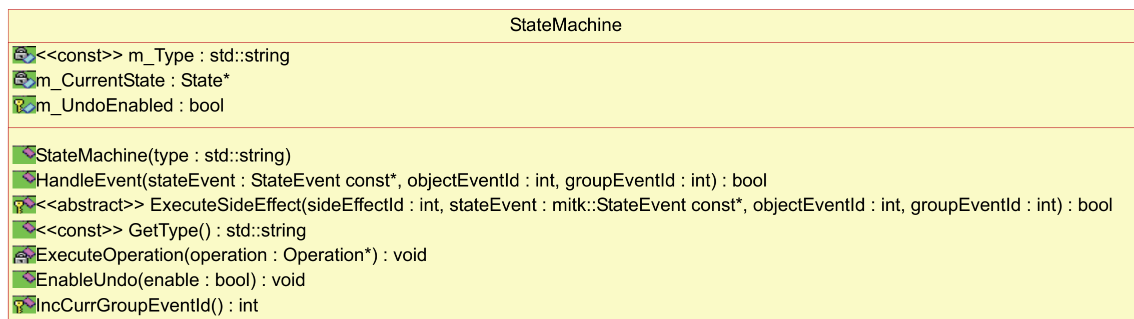


Abbildung 3.1: Klassen-Diagramm StateMachine

Eine Zustandsmaschine wird durch eine Klasse `StateMachine` (vgl. Abb. 3.1) realisiert. Ein Objekt dieser Klasse enthält einen Zeiger, der auf den aktuellen Zustand verweist. Ferner enthält es Methoden, die ein Ereignis auf Akzeptanz überprüfen und gegebenenfalls eine Aktion ausführen. Die Modellierung des Verhaltens bedient sich zweier Klassen, `State` und `Transition` (vgl. 3.2.2).

Der Ablauf (vgl. Abb. 3.2): Wenn ein Ereignis dem Objekt vom Typ `StateMachine` übergeben wird, so wird in einer Methode `HandleEvent(..)` überprüft, ob das Ereignis

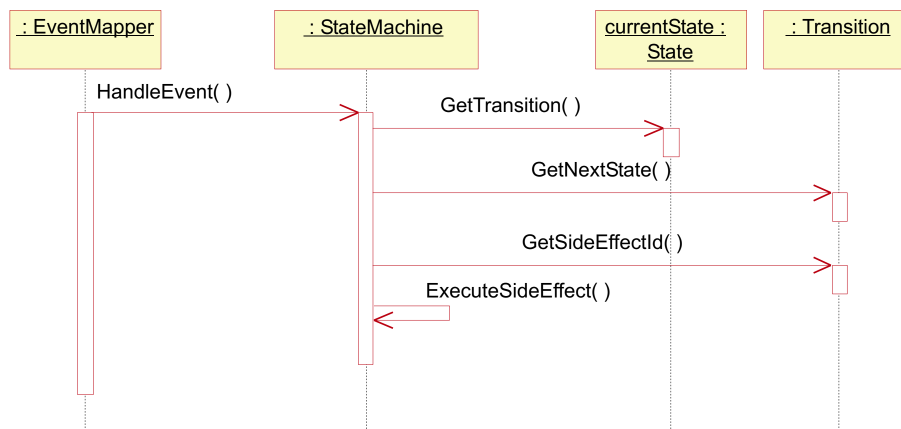


Abbildung 3.2: Sequenz-Diagramm Zustandsmaschine

in dem aktuellen Zustand akzeptiert wird. Es wird akzeptiert, wenn zu dem Ereignis ein Übergang definiert ist. Ist dies nicht der Fall, so wird die Methode mit einem Rückgabewert `false` beendet. Wird das Ereignis akzeptiert, so wird durch die Information im Übergang der Zeiger des aktuellen Zustands auf den resultierenden Zustand gesetzt. Hierauf wird von dem gleichen Übergang die definierte Aktion erfragt, die im Anschluss ausgeführt wird. Die Methode `HandleEvent(..)` endet mit dem Rückgabewert `true`.

Die Methoden der Klasse `StateMachine` realisieren einzig den Zustandswechsel. Von der Klasse abgeleitete Klassen müssen die zugehörigen Methoden zur Ausführung der Aktionen ergänzen (vgl. 3.2.8).

3.2.2 Zustand und Übergang

Das Verhalten einer Zustandsmaschine wird durch zwei Klassen abgebildet. Das steigert zum einen die Übersichtlichkeit des Quelltextes, zum anderen auch die der Definition aller Zustandsmaschinen.

Zustand

Die erste der zwei Klassen ist die Klasse `State`, welche einen Zustand darstellt (vgl. Abb. 3.3). Sie besitzt eine eindeutige Identifikationsnummer `m_Id`. In verschiedenen Maschinen können Zustände mit gleicher ID definiert sein. Jedoch ist es nicht zulässig, innerhalb einer Maschine zwei Zustände mit gleicher ID zu definieren.

Zum Zweck der Übersichtlichkeit ist ein Objekt der Klasse `State` mit einem Namen `m_Name` ausgestattet, der vom Entwickler frei gewählt werden kann. Ferner besitzt das Objekt eine Liste von Übergängen `m_Transitions`, die im Anschluss erklärt werden.

Die Methoden, die die Klasse `State` zur Verfügung stellt, setzen sich neben Konstruktoren und Destruktoren aus den üblichen Get-Methoden (`GetTransition(..)` etc.), Methoden

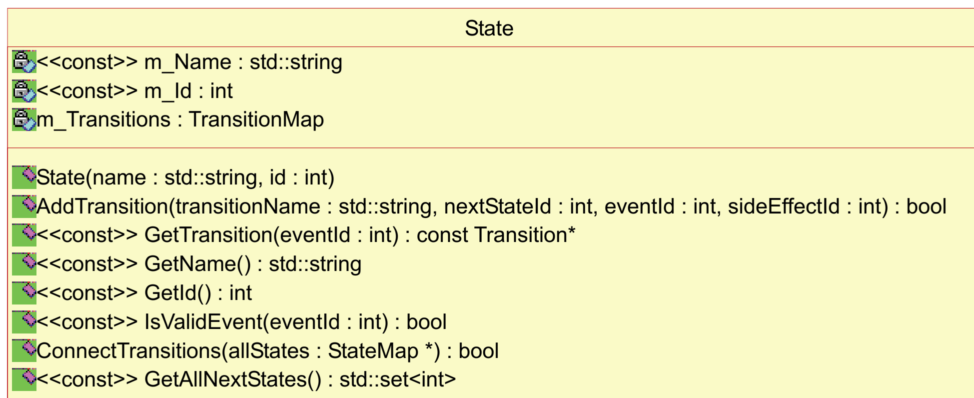


Abbildung 3.3: Klassen-Diagramm State

zur Überprüfung (`IsValidEvent(..)`) und Methoden, die für den Aufbau der Zustandsmaschine benötigt werden (`AddTransition(..)`, `ConnectTransitions(..)` etc.), zusammen.

Übergang

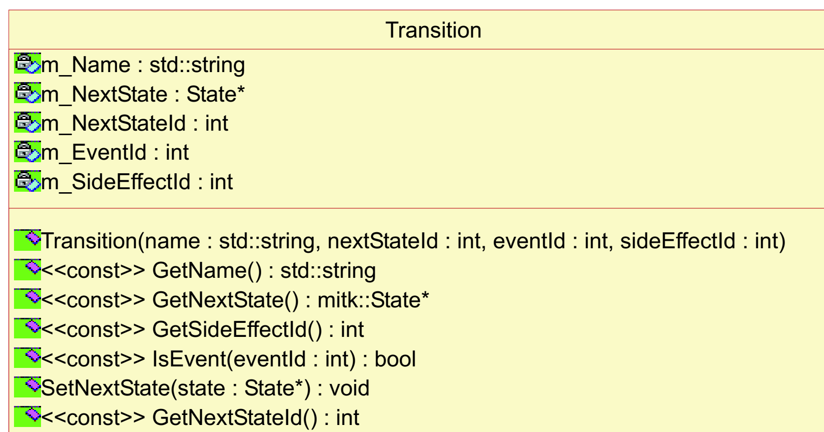


Abbildung 3.4: Klassen-Diagramm Transition

Die zweite der beiden Klassen ist die Klasse `Transition`, welche einen Übergang von einem Zustand in den nächsten darstellt (vgl. Abb. 3.4). Wie ein Objekt `State` besitzt ein Objekt des Datentyps `Transition` zur besseren Übersicht einen frei wählbaren Namen `m_Name`. Mit einer Ereignisnummer `m_EventId` ist ein Übergang an ein bestimmtes Ereignis gebunden. Wie in 2.7.1 beschrieben, gehört zu einem Ereignis auch eine Aktion, die hier in der Variable `m_SideEffectId` abgelegt ist. Um den Aufbau der Zustandsmaschine zu realisieren, ist dem Objekt in der Membervariable `m_NextStateId` die Identifikationsnummer des resultierenden Zustands beigefügt. Nach dem Aufbau besitzt das Objekt einen Zeiger `m_NextState` auf den nächsten Zustand. So kann über die Verwendung von Zeigern von

einem Zustand in den nächsten gewechselt werden.

Die Methoden setzen sich wie bei der Klasse `State` neben Konstruktoren und Destruktoren aus den üblichen Get-Methoden (`GetNextState(..)` etc.), Methoden zur Überprüfung (`IsEvent(..)`) und Methoden, die für den Aufbau der Maschine benötigt werden (`SetNextState(..)`), zusammen.

3.2.3 Generierung

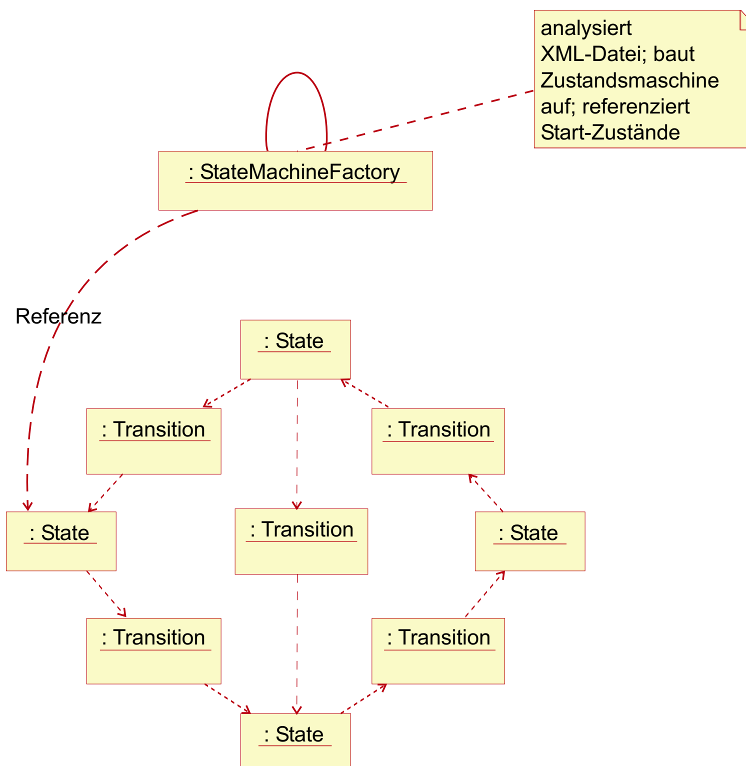


Abbildung 3.5: Objekt-Diagramm der Zustandsmaschine; Aufbau und Referenzierung des Start-Zustands durch ein Objekt der Klasse StateMachineFactory

Eine Zustandsmaschine wird zur Laufzeit aufgebaut. Die hierfür benötigten Definitionen über Zustände und Übergänge stehen geordnet in einer XML-Datei [Con03]. In einem Objekt der Klasse `StateMachineFactory` wird die Datei geöffnet, analysiert und die für den Aufbau der Zustandsmaschine nötigen Definitionen über Zustände und Übergänge herausgesucht. Aus den gesammelten Informationen wird im Anschluss die Zustandsmaschine aufgebaut. Hierfür werden Objekte der Datentypen `State` und `Transition` genutzt, durch deren Verbindung eine Zustandsmaschine entsteht (vgl. Abb. 3.5).

Nun ist in der XML-Datei nicht nur eine Zustandsmaschine definiert, sondern viele, die gemeinsam die komplexe Interaktion abbilden. So werden viele Zustandsmaschinen aufgebaut und jeweils über einen Zeiger auf den Start-Zustand in dem Objekt des Typs

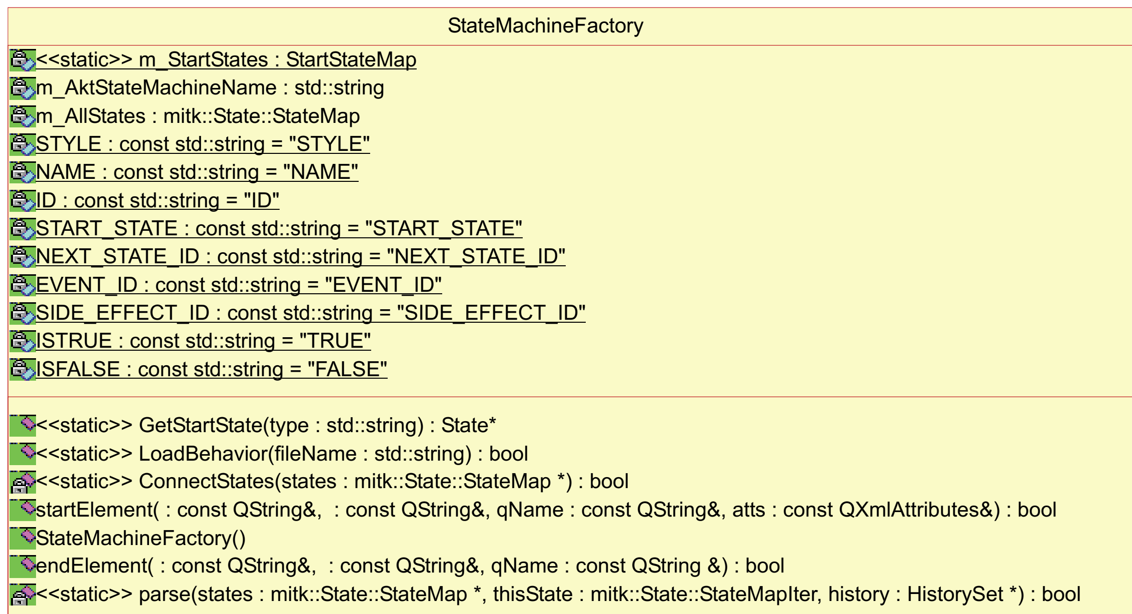


Abbildung 3.6: Klassen-Diagramm StateMachineFactory

`StateMachineFactory` referenziert. Über die Methode `GetStartState(..)` kann diese Referenz auf den initialen Zustand erfragt werden (vgl. Abb. 3.6).

3.2.4 Definition

Eine neue Zustandsmaschine soll möglichst leicht zu definieren sein. Sie sollte für die Interaktionsplanung zunächst grafisch erstellt werden, aber spätestens wenn zur Laufzeit alle Zustandsmaschinen aufgebaut werden sollen, muss die Definition in schriftlicher Form vorliegen.

Um die Übersichtlichkeit der Definitionen in der Schriftform so gut wie möglich zu wahren, wird auf den XML-Standard [Con03] zurückgegriffen. XML bildet Information in Ebenen ab, so dass die Definition der Zustandsmaschine übersichtlich organisiert werden kann.

Nach einer üblichen XML-Versionsnummer wird die erste Ebene mit dem Namen der Interaktions-Art eröffnet (vgl. Abb. 3.7). So kann nicht nur eine Art der Interaktion verwirklicht werden, sondern gleich mehrere. Eine Art der Interaktion könnte dem Verhalten der Präsentationssoftware Microsoft® PowerPoint®, eine andere dem des Grafikprogramms Adobe® PhotoShop® nachempfunden sein.

Die erste Ebene unterteilt sich in die zweite Ebene „Zustandsmaschine“, in der die Definition aller Maschinen dieser Interaktions-Art aufgelistet werden. Jede Maschine hat ihren eigenen Namen, der frei gewählt werden kann und der besseren Orientierung des Entwicklers dienen soll. Die Zustände der Maschine werden in der dritten Ebene definiert. Zum Schluss werden in der vierten Ebene die Übergänge aufgelistet.

Interaktions-Art PowerPoint

- **Zustandsmaschine** Global
  - **Zustand** 1
    - **Übergang** nach 2
    - **Übergang** nach 3
  - **Zustand** 2
    - **Übergang** nach 1
  - **Zustand** 3
    - **Übergang** nach 2
- **Zustandsmaschine** Punkt
  - **Zustand** 1
    - **Übergang** nach 2
  - **Zustand** 2
    - **Übergang** nach 1

Interaktions-Art PhotoShop

- **Zustandsmaschine** global
  - **Zustand** 1 ...

Abbildung 3.7: Hierarchischer Aufbau der Zustandsmaschinen in der XML-Datei

Die jeweiligen Informationen zum Zustand oder Übergang (Name, ID, Ereignis etc.) werden in die Definition des jeweiligen Elements mit eingebaut.

Ein Beispiel für den Aufbau einer XML-Datei zur Konfiguration von Zustandsmaschinen wird in 4.2.5 gezeigt.

3.2.5 Wiederverwendung der Interaktionsmuster

Um nicht für jedes zur Laufzeit erstellte Objekt mit Verhalten eine Zustandsmaschine erstellen zu müssen, wird ein ökonomisches Verfahren benötigt, in dem sich Objekte mit gleichem Verhalten ein *Verhaltensmuster* teilen. Dieses Verfahren schont Speicherplatz und Entwicklungszeit.

Die Umsetzung sieht eine Trennung des vorgestellten Aufbaus einer Zustandsmaschine vor: Die *Logik* der Maschine wird von ihrem *Verhaltensmuster* getrennt (vgl. Abb. 3.8). Zur Logik, hier auch Interactor genannt, zählen zum einen die Methoden, die ein Ereignis verarbeiten sowie gegebenenfalls Aktionen ausführen, zum anderen der aktuelle Zustand, der über einen Zeiger gehalten wird. Das Verhaltensmuster (Interaction-Pattern) beinhaltet alle aufgebauten `State`- und `Transition` -Objekte. Diese stehen untereinander in Verbindung und bilden so das Verhalten ab.

Kombinationen aus Logik und Muster können nun, ohne Veränderungen vorzunehmen,

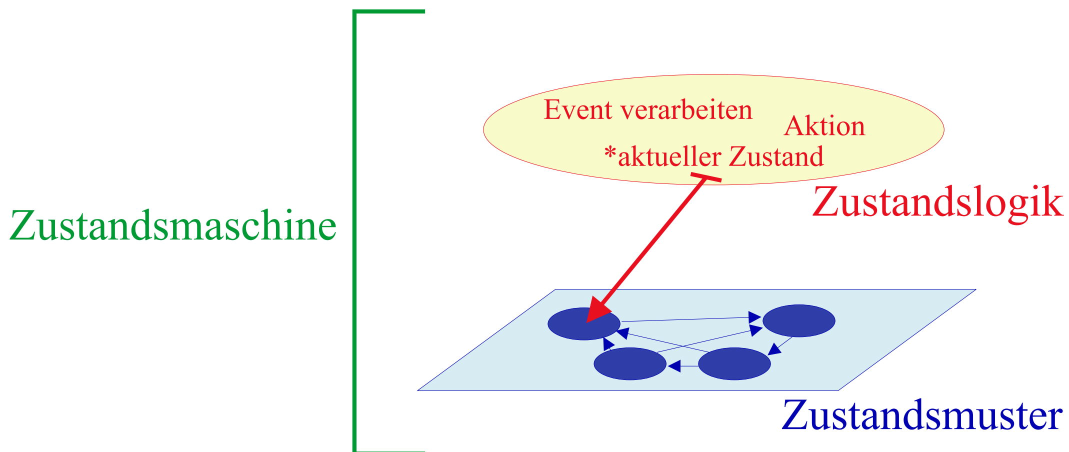


Abbildung 3.8: Teilung einer Zustandsmaschine in Logik und Verhaltensmuster

frei gebildet werden. Dadurch ist es auch möglich, dass ein Muster von mehreren Logiken genutzt werden kann (vgl. Abb. 3.9). So werden der Speicherplatz vieler gleicher Muster und die Zeit, die der Aufbau der Muster benötigt, gespart.

Der gemeinsame Gebrauch von Logik bietet diese Vorteile nicht. Das Verhalten über die Zeit wird vom aktuellen Zustand bestimmt. Dieser wird in der Logik referenziert. Würden sich zwei Objekte eine Logik teilen, so müssten sie sich auch einen aktuellen Zustand teilen. Veränderungen des einen Objekts würden sich auf das andere auswirken. Da das nicht im Sinne einer prädiktiven Programmlogik ist, wird in diesem Konzept darauf verzichtet.

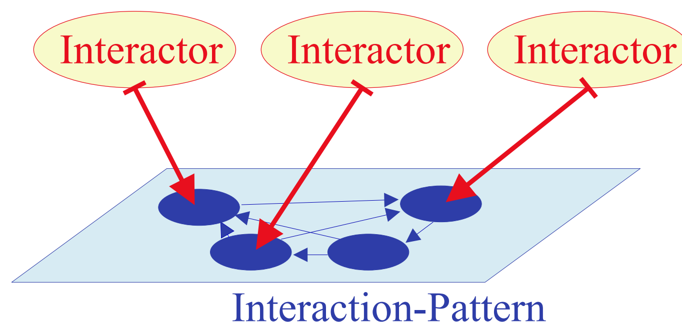


Abbildung 3.9: Zugriff mehrerer Zustandslogiken auf ein Zustandsmuster

3.2.6 Hierarchie

Um die gesamte komplexe Interaktion des Programms für den Entwickler übersichtlich zu gestalten, können wir sie in kleinere Teil-Interaktionen zerlegen.

Betrachten wir eine mögliche Hierarchiebildung an einem Beispiel (vgl. Abb. 3.10): Ein Programm mit vier Ansichten wird gestartet. Im laufenden Betrieb hat der Benutzer die

Global	Fokus der Ansichten
Lokal	Zoom / Pan einer Ansicht
Werkzeug	Parameter einstellen
Objekt	verschieben / löschen
Teilobjekt	verschieben /löschen
Linie	erstellen /verändern
Punkt	setzen, löschen

Abbildung 3.10: Beispiel einer Hierarchiebildung von Interaktionen

Möglichkeit, auf jeweils eine Ansicht zu interagieren. Hier können wir unsere erste Ebene definieren. Eine Zustandsmaschine mit entsprechendem Verhalten kümmert sich um einen Fokus auf eine der vier Ansichten. Eine Ansicht hat ein bestimmtes Verhalten. Der Benutzer kann in ihr zoomen, den Blickwinkel ändern, verschieben etc. In unserem Beispiel stellt das die zweite Ebene dar. In der dritten Ebene können wir ein von der Ansicht abhängendes Werkzeug definieren. Es erstellt nach eingegebenen Parametern eine Objektumgebung. Die vierte Ebene wird von einer Zustandsmaschine verwaltet, die sich um das Verschieben oder das Löschen eines Objekts kümmert. Dieses Objekt kann aus mehreren Teilobjekten zusammengesetzt sein. Nehmen wir in diesem Beispiel einen Kurven- und einen Linienverlauf (vgl. Abb. 2.10). Jedes Teilobjekt hat ein spezifisches Verhalten, das wir durch eine entsprechende Zustandsmaschine realisieren. So kann der Benutzer den gesamten Kurvenverlauf verschieben, löschen oder ähnlich verändern. Diese Zustandsmaschine stellt unsere fünfte Ebene in der Hierarchie dar. Ein solches Teilobjekt wiederum kann aus mehreren Elementen bestehen, z.B. der Linienverlauf aus mehreren Linien. In unserer Abbildung 3.10 ist diese Maschine in der vorletzten Ebene dargestellt. Die letzte Ebene unserer Hierarchie nimmt ein Punkt ein, dessen Verhalten wir wohl in den meisten (Teil-)Objekten gebrauchen können. Einen Punkt kann man setzen, selektieren, deselektieren, verschieben und wieder löschen.

Der Benutzer erfährt die Interaktion als Ganzes und merkt nicht, dass sie aus Komponenten zusammengesetzt wird. Der Entwickler jedoch profitiert von der Teilung der Interaktion, da dadurch dem Verhalten die Komplexität genommen wird. Entwicklungen, Veränderun-

gen und Anpassungen können so vereinfacht und beschleunigt werden.

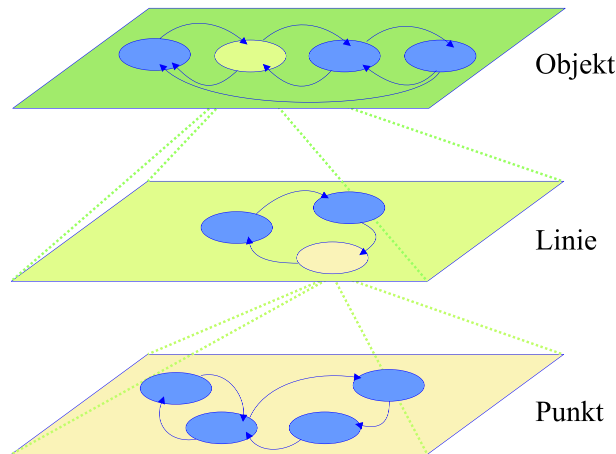


Abbildung 3.11: Modellierung der Interaktion in Abstraktionsebenen

Das Konzept realisiert die Unterteilung in Ebenen, indem zugelassen wird, dass eine Zustandsmaschine weitere Zustandsmaschinen unterer Hierarchie-Ebenen referenzieren darf. Es darf nur in einer Richtung referenziert werden: von oben nach unten (vgl Abb. 3.10).

Der Entwickler kann so bei der Erstellung eines neuen Werkzeugs auf schon bestehende Zustandsmaschinen zurückgreifen (Wiederverwendung) und braucht sich so nur auf die Modellierung des neuen Verhaltens zu konzentrieren. Dabei kann dies auf einer *Abstraktionsebene* geschehen. Eine solche Ebene schließt das Verhalten einer Zustandsmaschine mit ein und fasst das Verhalten der in der Hierarchie tiefer liegenden Maschinen in einem Zustand zusammen (vgl. Abb. 3.11). Durch die Hierarchiebildung verringert sich die Anzahl der Zustände einer Abstraktionsebene. Somit können die einzelnen Ebenen eines neuen Werkzeugs parallel entwickelt werden.

3.2.7 Aktion

Wie wir schon in 2.7.1 erfahren haben, ist jedem Übergang eine Aktion zugeordnet. In dem hier vorgestellten Konzept wird dies über einen Parameter `SideEffectId` realisiert, der in dem Objekt `Transition` gespeichert wird. Wird ein Ereignis an die Zustandsmaschine geschickt, wird überprüft, ob in dem aktuellen Zustand das Ereignis akzeptiert werden kann. Dies geschieht, wenn in diesem Zustand ein Übergang definiert ist, dessen Parameter `EventId` mit dem Ereignis übereinstimmt. Wird das Ereignis akzeptiert, so wird der Zeiger des aktuellen Zustands auf den resultierenden Zustand versetzt und im Anschluss die Methode `ExecuteSideEffect(..)` aufgerufen. Die Information, die in der Variable `SideEffectId` steht, wird mit übergeben. In der Methode wird dann überprüft, ob für diese `SideEffectId` ein Anweisungsblock implementiert wurde. Ist dies der Fall, so wird dieser ausgeführt. Ist

dies nicht der Fall, so wird die Aktion als Null-Aktion (vgl. 2.7.4) verstanden und entsprechend einer Default-Anweisung behandelt. In einem Anweisungsblock können beispielsweise Daten wie Punktkoordinaten verändert, Selektionen von Punkten vorgenommen oder Filter auf einen Bildbereich angewandt werden. Jedoch können in einem Anweisungsblock auch Bedingungen überprüft werden. Je nach Ergebnis der Überprüfung kann ein neues Ereignis generiert werden, welches dann wie ein vom Benutzer erzeugtes Ereignis verarbeitet wird. So ist es möglich, die in 2.7.6 besprochenen *Wächter* zu realisieren. Ferner kann in der Methode entweder das empfangene oder ein neu generiertes Ereignis an die in der Hierarchie untergeordneten Zustandsmaschinen übergeben werden. Diese können das Ereignis dann für sich verarbeiten.

3.2.8 Vererbung der Zustandsmaschinen

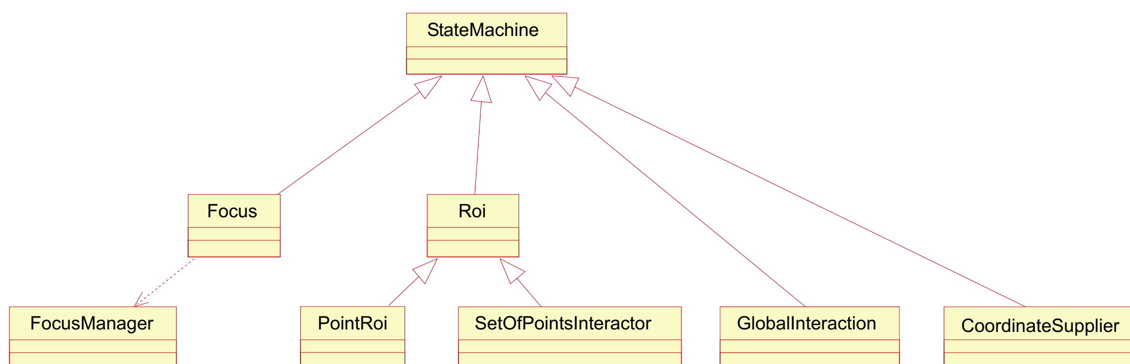


Abbildung 3.12: Übersicht der von StateMachine abgeleiteten Klassen

In einem solch großen Projekt wie dem MITK ist es wichtig, den Entwickler bei der Implementierung neuer Applikationen bestmöglich zu unterstützen. Bereits realisierte Entwicklungen sollten, so weit es geht, genutzt werden. Hierfür steht dem Entwickler durch die objektorientierte Programmiertechnik die Möglichkeit der Ableitung einer Klasse zur Verfügung.

Wenn eine bereits implementierte Klasse `A` einen Teil der zu realisierenden Aufgabe erfüllt, kann von dieser eine neue Klasse `B` abgeleitet werden. Methoden und Membervariablen, die von `A` *geerbt* wurden, können von `B` wie die eigenen genutzt werden. Zusätzlich können noch weitere Methoden oder Membervariablen in `B` deklariert werden. Wenn eine von `A` geerbte Methode von `B` in veränderter Form benötigt wird, kann diese Methode in `B` *überschrieben* werden.

Durch die Möglichkeit der Vererbung kann die Klasse `StateMachine` ohne sonderliche Doppelimplementierung vielseitig genutzt werden. Abgeleitete Klassen verhalten sich von außen betrachtet wie komplett neu erstellte Klassen, mit dem Unterschied, dass bei der Im-

plementierung wertvolle Entwicklungszeit gespart wurde. Wie in Abbildung 3.12 dargestellt, kann das Konstrukt der Zustandsmaschine für das Verhalten eines Werkzeugs (ROI) herangezogen, aber auch für die Verwaltung mehrerer Ansichten (Fokus) genutzt werden. Das Objekt der Klasse `GlobalInteraction`, das an oberster Stelle der Interaktionshierarchie steht, kümmert sich global um die Interaktion und leitet einzelne Ereignisse an untergeordnete Maschinen weiter. Während der Erstellung der ersten Applikationen haben sich weitere Objekte herauskristallisiert, die ebenfalls ein spezielles Verhalten benötigen.

Alle Klassen der Art Zustandsmaschine, die für die direkte Interaktion auf Daten zuständig sind (im Folgenden auch *Interactor* genannt), werden von der Klasse `ROI` abgeleitet. In dieser Klasse sind alle Methoden enthalten, die von allen Interactor-Objekten gebraucht werden. In der Abbildung 3.12 ist dies durch die beiden Klassen `SetOfPointsInteractor` und `PointRoi` dargestellt.

Alle Zustandsmaschinen, die ein Verhalten abbilden und gegebenenfalls Aktionen ausführen, werden von der Klasse `StateMachine` abgeleitet. In `StateMachine` wird lediglich der Zustandswechsel realisiert, die abgeleiteten Klassen jedoch realisieren die Ausführung der Aktionen. Dabei können auch von einer bereits abgeleiteten Klasse weitere Klassen abgeleitet werden. Durch einen entsprechenden Aufruf kann so die Bearbeitung von Aktionen übernommen und Bearbeitung neuer Aktionen hinzugefügt werden.

3.2.9 Null-Übergang

Während des ersten Tests mit hierarchischem Aufbau der Zustandsmaschinen stellte sich Folgendes heraus: Die Anzahl der Übergänge in Zuständen oberer Hierarchie-Ebenen steigt mit der Komplexität der untergeordneten Maschinen. In der Zustandsmaschine wird nach dem Empfang eines Ereignisses überprüft, ob in dem aktuellen Zustand ein Übergang existiert, der eine entsprechende `m_EventId` (Ereignis-Identifikationsnummer) enthält. Ist dies der Fall, so wird der aktuelle Zustand gewechselt und `ExecuteSideEffect(..)` aufgerufen. Ist dies nicht der Fall, wird das Ereignis ignoriert, kein Zustandswechsel vorgenommen und auch keine Aktion ausgeführt.

Wird das Ereignis nicht akzeptiert, kann es auch nicht an untergeordnete Maschinen gesendet werden, da die Weiterleitung in einer Aktion durchgeführt wird. Damit es weitergeleitet werden kann, müssen zwangsläufig in den oberen Maschinen in einem Zustand alle Ereignisse der unteren Maschinen definiert sein.

Da das nicht im Sinne der Übersichtlichkeit ist, wurde eine neue Art des Übergangs definiert, der *Null-Übergang*. Durch diesen Übergang wird das Problem behoben, denn in diesem werden alle Übergangs-Deklarationen unterer Maschinen komprimiert. Ist ein solcher Übergang in einem Zustand definiert, wird in der Zustandsmaschine das Ereignis nicht ignoriert, sondern eine Aktion aufgerufen, in der dann das Ereignis an die unteren Maschinen weitergegeben werden kann.

Ein Null-Übergang wird, wie alle anderen Übergänge, durch ein Objekt `Transition` realisiert, wird aber durch eine `m_EventId` = 0 gekennzeichnet. Alle übrigen Übergänge beinhalten eine `m_EventId` > 0. Ein Null-Übergang kann eine beliebige `m_NextStateId` oder `m_SideEffectId` enthalten. Der Name `m_Name` sollte vom Entwickler so gewählt werden, dass aus ihm geschlossen werden kann, dass es sich um einen Null-Übergang handelt und das Ereignis weitergeleitet wird.

3.2.10 Umgang mit Ereignissen

Zu der Interaktion gehören nicht nur die Zustandsmaschinen, das Verhalten von Objekten oder die Funktionalität eines Undos, sondern auch die Verarbeitung von Eingaben, im Folgenden als Ereignisse (Event) beschrieben. Ohne diese Ereignisse wäre der Benutzer nicht in der Lage auf die Objekte zu interagieren. Was sozusagen die Schallwellen für die lautsprachliche Kommunikation zweier Menschen sind, ist ein Ereignis für die Interaktion zwischen Mensch und Computer.

Die Anforderung, schnell Änderungen an der Interaktion vorzunehmen, muss nicht nur im Konzept der Zustandsmaschinen bedacht werden, sondern auch in dem der Ereignisse. Definitionen können dann schnell geändert werden, wenn sie an zentraler Stelle stehen und nicht im Quellcode in den einzelnen Klassen deklariert werden. Wie die Zustandsmaschinen werden deshalb alle Ereignisse, die von den Maschinen verarbeitet werden sollen, in der schon angesprochenen XML-Datei definiert. Die Syntax der Definition ist hierbei der der Zustandsmaschinen ähnlich (vgl. Abb. 3.13).

Ereignis-Art PowerPoint

- **Ereignis** LeftMouseButton
- **Ereignis** MiddleMouseButton
- **Ereignis** RightMouseButton
- **Ereignis** Key A
- **Ereignis** Key Del
- **Ereignis** Key Return ...

Abbildung 3.13: Definition der Ereignisse in der XML-Datei

Ereignisse werden zwar vom Benutzer erzeugt, jedoch vom Betriebssystem bzw. in diesem Falle von QT, welches für die Modellierung der Benutzeroberfläche genutzt wird [Tro03a] [Dah99], an das Toolkit MITK übermittelt. QT hat ein eigenes Konzept Ereignisse zu verarbeiten und zu kodieren. Unterstützt werden von QT die gängigen Eingabegeräte wie Maus, Tastatur oder Grafiktablett. Das Toolkit MITK soll jedoch nicht nur an die beschriebenen Eingabegeräte gebunden sein, sondern auch besondere Eingabegeräte unterstützen. Speziell

in der Medizin werden des Öfteren außergewöhnliche, eigens entwickelte Eingabegeräte zur Interaktion genutzt, da es z.B. im Operationssaal nicht möglich ist, Tastatur und Maus zur Hand zu nehmen. Deswegen muss die Interaktions-Komponente des MITKs anpassungsfähig an neue Eingabegeräte sein. Zum einen wird dies durch den schon beschriebenen Einsatz der XML-Datei realisiert und zum anderen werden die Vergleiche im Quellcode über Konstanten vorgenommen. Eine weitere Datei beinhaltet alle Konstanten-Definitionen, so dass dem Entwickler Konstanten zur Verfügung stehen, mit denen er unabhängig von QT oder anderen Bibliotheken mit Ereignissen arbeiten kann. Zusätzlich der Anwendung von QT- auf MITK-Konstanten werden hier auch Konstanten für Aktions-IDs, Operations-IDs etc. definiert. Die Konstanten-Datei übernimmt nicht nur die Aufgabe, Konstanten zu definieren, sondern verbindet auch den Quellcode mit der XML-Datei. Die Definition der Zustandsmaschine soll so wenig Ziffern wie möglich beinhalten. Identifikationsnummern müssen natürlich in Ziffern geschrieben werden, jedoch können Angaben für die Aktion oder das Ereignis auch durch Konstanten, hinter denen sich jeweils eine Ziffer verbirgt, übersichtlicher beschrieben werden. Gleiches gilt für die Definition der Ereignisse. Hier sollten auch so wenig Ziffern wie möglich gebraucht werden.

In einer Definition eines Ereignisses in der XML-Datei sind folgende Elemente enthalten:

Name frei wählbar zur besseren Übersicht

ID interne MITK-Ereignis-ID

Type Art des Ereignisses (MouseButtonPress, MouseMove, KeyPress, TabletPress etc.)

Button evtl. auslösende Taste der Maus

ButtonState Modifikator (Shift, Control, Meta etc.)

Key evtl. auslösende Taste der Tastatur

Betrachten wir den Aufbau genauer, so sehen wir, dass die Informationen der Tastatur und der Maus in einer Definition enthalten sind. Die GUI-Bibliothek QT erstellt hierfür zwei unterschiedliche Objekte `QKeyEvent` und `QMouseEvent` [Tro03b]. Auf die Unterteilung in Tastatur- und Maus-Ereignis wurde bei dem generischen Interaktionskonzept des MITKs verzichtet, damit auch Kombinationen aus Tastatur und Maus möglich sind.

Ein Beispiel: Der Benutzer hält „ATL“ zusammen mit „A“ gedrückt. Nun setzt er bestimmte Punkte in der Ansicht mit der linken Maus-Taste.

Die Kombination der Eingaben kann in einem Ereignis zusammengefasst werden. In der Zustandsmaschine kann das spezielle Ereignis „ALT+A+LeftMouseButton“ einen Übergang auslösen, der eine Aktion zur Folge hat. Natürlich kann die Kombination, Tastatur und Maus, auch über zwei Ereignisse realisiert werden. Ein Übergang, der bei dem Ereignis

„ALT + A“ ausgelöst wird, gefolgt von einem Übergang, der auf die linke Maus-Taste reagiert, führt zu dem gleichen Ergebnis.

Ebenso können die Eingaben eines speziellen Eingabegeräts mit denen der Standardeingabegeräte, Maus und Tastatur, kombiniert werden.

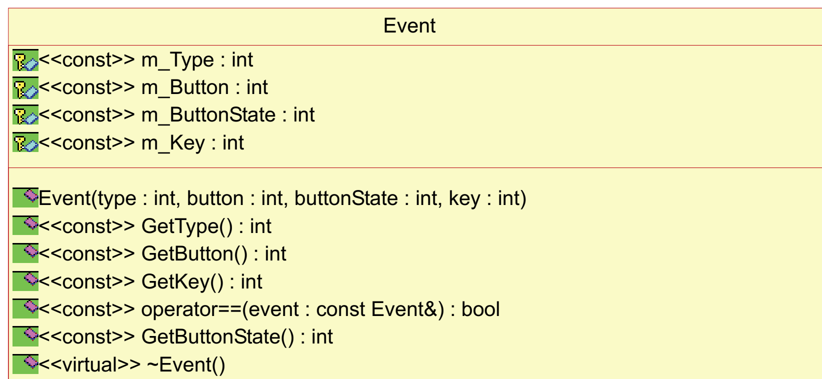


Abbildung 3.14: Klassen-Diagramm Event

Das Objekt `Event` beinhaltet die Membervariablen `m_Type`, `m_Button`, `m_ButtonState` und `m_Key` (vgl. Abb. 3.14). Werden weitere Informationen für die Verarbeitung eines Ereignisses benötigt, so werden diese in einem von `Event` abgeleiteten Objekt übermittelt. Abbildung 3.15 zeigt ein Diagramm der Klasse `PositionEvent`. Die Klasse fügt die Information der Position des Mauszeigers hinzu.

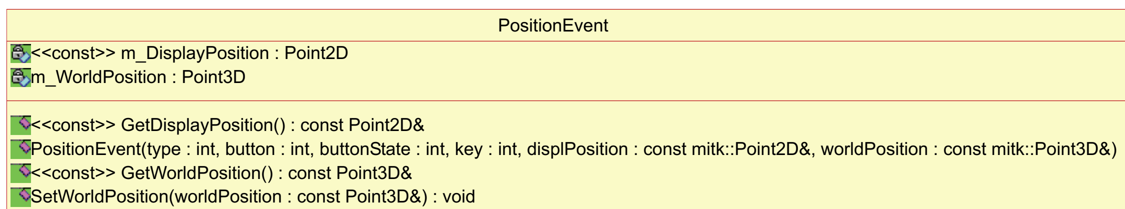


Abbildung 3.15: Klassen-Diagramm PositionEvent

Besonders ist, dass in der Klasse nicht nur die zweidimensionalen Koordinaten der Position des Mauszeigers, sondern auch die dritte Koordinate für die Tiefe gespeichert wird. Der letzte Parameter wird in den Visualisierungs-Objekten des MITKs errechnet (Picking) und dient der dreidimensionalen Interaktion. Diese Koordinaten stehen dann allen Interaktions-Objekten (*Interaktoren*) des MITKs zur Verfügung und werden nicht nur speziell für 3D-Interaktion berechnet. Ferner ist somit der Einsatz von dreidimensionalen Eingabegeräten vorbereitet. Ein Phantom [MS94] [OPB+97] oder ein Tracking-System [VWH+03] können somit schnell in die MITK-Interaktion integriert werden.

Wie die Klasse `PositionEvent` können noch weitere Klassen, die für eine bestimmte

Verarbeitung nötig sind, von der Klasse `Event` abgeleitet werden.

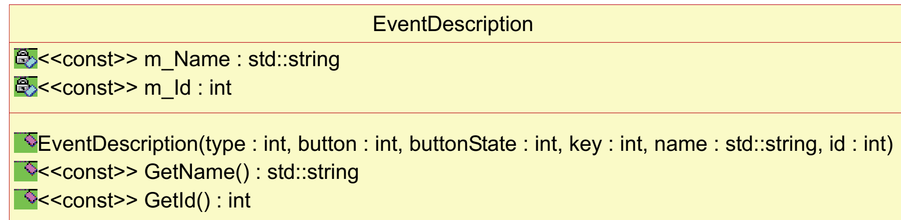


Abbildung 3.16: Klassen-Diagramm EventDescription

Die Parameter Name und ID, die in der XML-Datei definiert, jedoch nicht in `Event` gespeichert werden, werden in einem Objekt `EventDescription` zusammen mit dem Objekt `Event` gespeichert (vgl. Abb. 3.16). `EventDescription` stellt ein internes MITK-Event dar. Hier verbindet sich so langsam die Beschreibung des Konstrukts Zustandsmaschine mit dem des Ereignisses:

$$\texttt{m_Id}_{Ereignis} = \texttt{m_EventId}_{Zustandsmaschine}$$

Die Identifikationsnummer `m_Id`, die in `EventDescription` an ein Objekt `Event` gebunden wird, wird für einen Zustandswechsel einer Zustandsmaschine benötigt. Im Folgenden gehen wir darauf ein, wie ein Ereignis an die Zustandsmaschine übermittelt wird. Mit den bisher beschriebenen Informationen über Ereignisse und ihre Repräsentation im MITK können wir nun den Ablauf untersuchen.

Die initiale Verarbeitung von externen Ereignissen übernimmt eine spezielle Klasse, deren Aufbau von dem Eingabegerät oder von den zu konvertierenden Ereignissen abhängt. Ereignisse, die z.B. von QT an die Klassen des MITKs gesendet werden, müssen in MITK-Ereignisse des Typs `Event` umgeschrieben werden. Werden zusätzliche Variablen benötigt, so kann eine Klasse von `Event` abgeleitet und angepasst werden. Hierdurch ist die Verarbeitung von Ereignissen von keiner Bibliothek oder keinem Eingabegerät abhängig. Die konvertierten MITK-Ereignisse werden an ein zentrales Objekt des Typs `EventMapper` in der Methode `MapEvent(..)` übergeben (vgl. Abb. 3.17).

Das Objekt analysiert zur Laufzeit, ähnlich wie das Objekt `StateMachineFactory`, die XML-Definitions-Datei und schreibt die gefundenen Definitionen in neu erstellte Objekte vom Typ `EventDescription`. Die hier definierten Beschreibungen werden in einer internen Liste aufgenommen.

Bei einem Aufruf der Methode `EventMapper::MapEvent(Event *event)` wird überprüft, ob sich in der internen Liste zu dem übergebenen Parameter `event` eine Beschreibung, also ein Objekt des Typs `EventDescription`, finden lässt. Ist dies nicht der Fall, so wird das Ereignis ignoriert. Wird eine Beschreibung gefunden, so wird diese zusammen mit dem zuvor empfangenen Objekt `event` in die statische Membervariable `m_StateEvent` geschrieben (vgl.

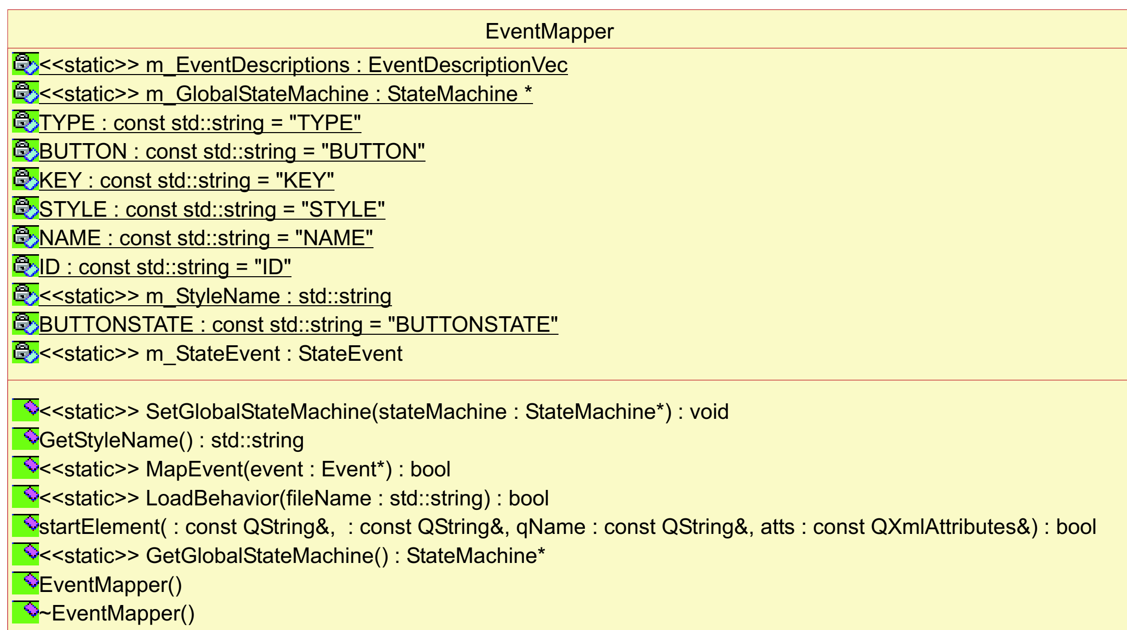


Abbildung 3.17: Klassen-Diagramm EventMapper

Abb. 3.18). Durch die statische Eigenschaft der Variable ist gewährleistet, dass immer nur ein Ereignis verarbeitet wird und es so nicht zu Überschneidungen kommt. Zudem werden Ressourcen gespart.

In dem Objekt `m_StateEvent` werden `m_Id` und `m_Event` gespeichert. Das Objekt wird an die globale Zustandsmaschine übergeben, in der nachgeschaut wird, ob es zu `m_Id` einen Übergang gibt. Das Ereignis wird dann, wie in 3.2.1 und Abb. 3.2 beschrieben, weiterverarbeitet.

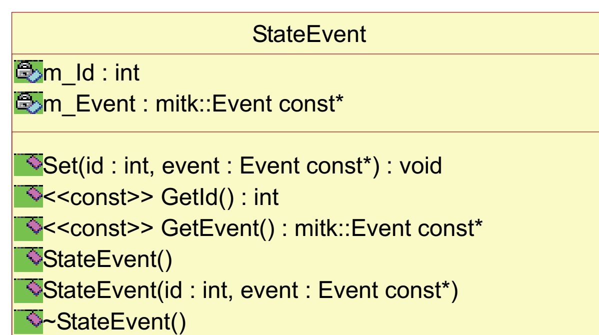


Abbildung 3.18: Klassen-Diagramm StateEvent

3.2.11 Zuordnung von Interaktions-Objekten zu Daten

Durch die hierarchische Anordnung von Zustandsmaschinen ist es möglich, Interaktion auf einem Abstraktionslevel zu modellieren. Auf diesem Level bezieht sich ein Interactor auf ein Daten-Objekt, z.B. auf eine Liste von Punkten. Die Zuordnung des Interactors zu dem Daten-Objekt wird in einem MITK-*Datenbaum* vorgenommen. Der semantisch geordnete Baum beinhaltet alle für die Verarbeitung wichtigen Daten, Visualisierungskomponenten (Mapper) und Interaktions-Objekte (StateMachine). Dabei sind zusammengehörige Komponenten in einem Knoten (Node) zusammengefasst (vgl. Abb. 3.19). In jedem Knoten sind Eigenschaften gespeichert, in denen beispielsweise die Selektion oder Farbe eines Punktes abgelegt werden kann.

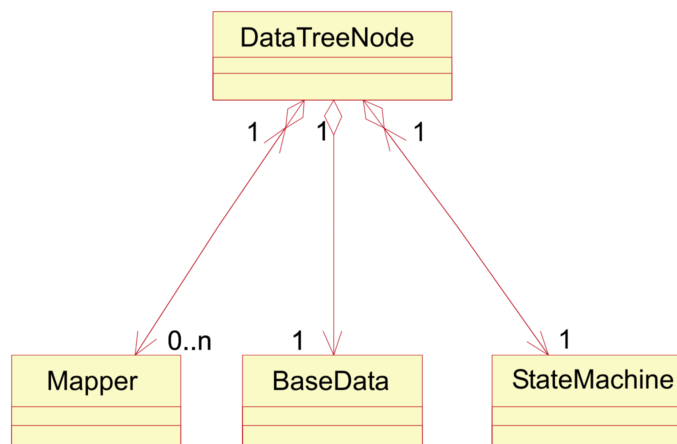


Abbildung 3.19: Klassen-Diagramm eines Datenbaum-Knotens

Durch die Zuordnung im Knoten „kennt“ eine Zustandsmaschine auch ihr zu verwaltendes Datum. Wird nun eine Aktion in einer Zustandsmaschine ausgeführt, wird über den Knoten auf das Datum zugegriffen und dies entsprechend verändert.

Zustandsmaschinen können jedoch auch an anderer Stelle eingesetzt werden. Eine Zustandsmaschine `GlobalInteraction`, die sich global um das Verhalten einer Anwendung kümmert, verwaltet kein Datum, sondern eine Vielzahl von Zustandsmaschinen. Infolgedessen wäre sie im Datenbaum falsch untergebracht und besser in der Hauptanwendung aufgehoben.

3.2.12 Gesamtübersicht des Interaktions-Modells

Den Gesamtüberblick und zugleich eine Zusammenfassung aller wichtigen Interaktions-Klassen bietet die Abbildung 3.20. Die einzelnen Klassen sind nummeriert und werden im Folgenden kurz beschrieben.

1 QXMLDefaultHandler QT-Basis-Klasse zur Analyse eines XML-Dokuments

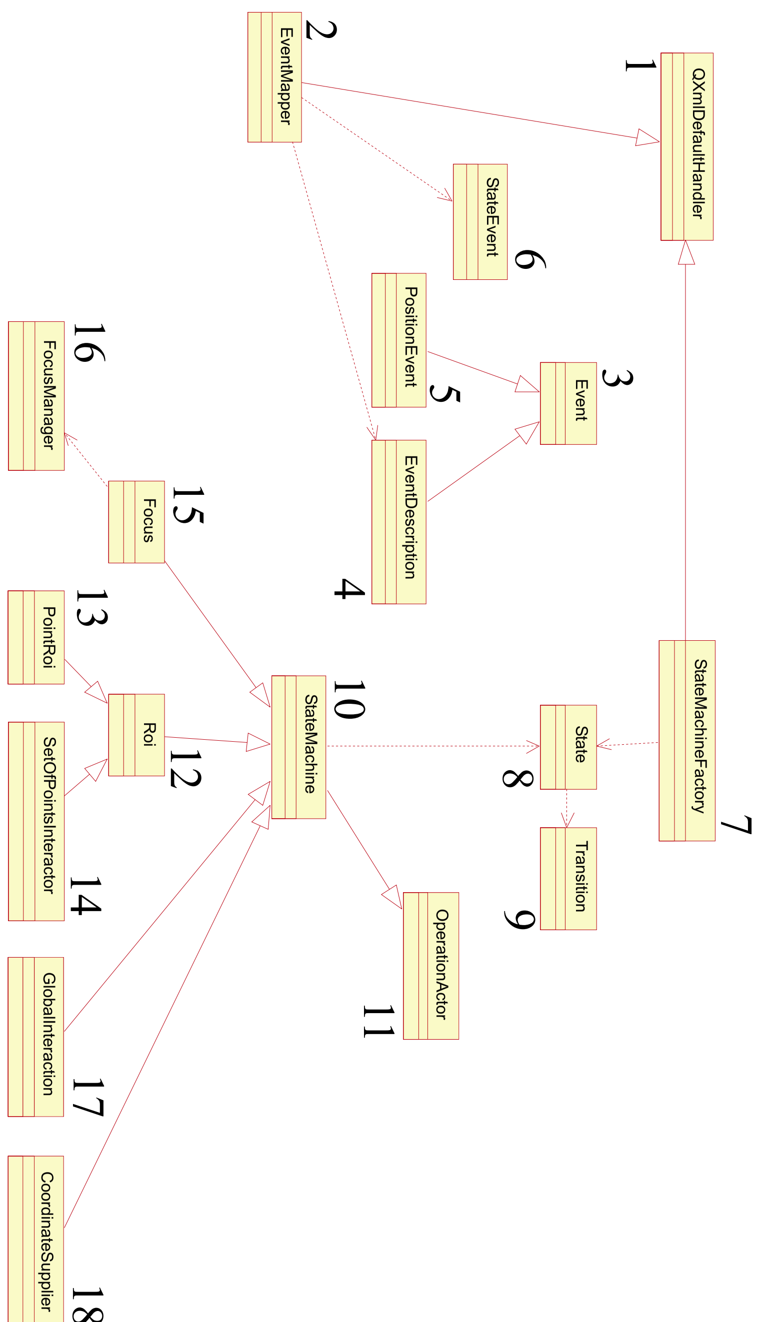


Abbildung 3.20: Übersicht der wichtigsten Interaktions-Klassen

2 EventMapper ordnet externen Ereignissen eine interne Identifikationsnummer zu

3 Event stellt ein Ereignis mit Informationen von Maus und Tastatur dar

4 EventDescription erweitert Event um Name und Identifikationsnummer für die Zuordnung im EventMapper

5 PositionEvent erweitert Event um eine dreidimensionale Koordinate

6 StateEvent fügt Ereignis-ID und Event zusammen; nötig für die Durchführung eines Zustandswechsels

7 StateMachineFactory baut die Zustandsmaschinen auf; referenziert alle Start-Zustände

8 State stellt einen Zustand dar

9 Transition stellt einen Übergang dar

10 StateMachine enthält die Logik einer Zustandsmaschine; referenziert einen Zustand in einem Verhaltensmuster

11 OperationActor Interface aller Daten-Klassen

12 ROI Region of Interest; Zustandsmaschine, die Methoden für weitere Interaktions-Maschinen bereitstellt

13 PointROI Zustandsmaschine, die das Verhalten eines Punktes abbildet

14 SetOfPointsInteractor Zustandsmaschine, die das Verhalten von einer Anzahl von Punkten abbildet

15 Focus Zustandsmaschine, die das Verhalten von verschiedenen Widgets (Trick-Fenster, Ansicht mit Interaktion) abbildet

16 FocusManager verwaltet eine Liste von Widgets; ein Element kann selektiert werden

17 GlobalInteraction Zustandsmaschine, die sich um die globale Interaktion kümmert; schickt Ereignisse an ausgewählte Maschinen weiter

18 CoordinateSupplier Zustandsmaschine, die nach einem bestimmten Verhalten Koordinaten weiterleitet

3.3 Entwurf des Undo-Konzepts

Die Wahl eines Undo-Modells fiel nicht leicht, da mit einem Toolkit wie dem MITK viele medizinische Anwendungen realisiert werden sollen und jede für sich speziell ist. Unterschiedliche Benutzerkreise erschweren die Auswahl. Daher wurde ein allgemeines Undo-Konzept entwickelt, mit dem mehrere Undo-Modelle realisiert werden können. Das hat zur Folge, dass Aufgaben, die in anderen Konzepten durch das Undo-Modell übernommen werden, vom Entwickler durchgeführt werden. Das allgemeine Konzept erlaubt eine vielseitige Verwendung der Undo-Funktionalität. Dem Entwickler werden mehrere Undo-Modelle angeboten, von denen er sich eines für die Erstellung seiner Anwendung aussuchen kann. Ebenso hat der Entwickler die Möglichkeit, die Auswahl der Modelle zu begrenzen und dem Benutzer die Wahl des Modells zu überlassen. Modelle können selbst während der Laufzeit gewechselt werden.

Die Strategie, die dem MITK-Undo-Konzept zugrunde liegt, ist die der *inversen Operation* (inverse command strategy, vgl. 2.8.5). Durch eine feingranuläre Wahl der Operationen wird gewährleistet, dass zu einer Operation eine inverse Operation gefunden werden kann. Ob die Inverse über Parameter oder die Art der Operation realisiert wird, ist dabei zweitrangig. Die Zuordnung der Operation zu ihrer inversen wird vom Entwickler vorgenommen, der die Interaktion und somit die Operationen zur Veränderung von Daten nach Wunsch des Benutzers implementiert. Somit übernimmt der Entwickler die Verantwortung, inverse Operationen korrekt zu definieren und Doppeldeutigkeiten von Inversen („Punkt_hinzufügen“, „Punkt_neu“; vgl. 2.8.5) zu beseitigen oder zumindest Inverse konsequent zuzuteilen.

Andere Strategien, wie z.B. die *Strategie des kompletten erneuten Ausführens* (vgl. 2.8.5), kamen für das Konzept nicht infrage, da sie eine zu langsame bzw. speicherintensive und somit benutzerunfreundliche Ausführungseigenschaft besitzen.

Eine Anforderung an eine benutzerfreundliche Umsetzung eines Undo-Modells ist die Transparenz des Verhaltens (vgl. 2.8.4). Das MITK-Undo-Konzept bietet dem Entwickler an, für die zu realisierende Anwendung das passende Undo-Modell auszuwählen. Besonders bei der Wahl eines komplexeren Undo-Modells muss er darauf achten, dass die Anwendung die Anforderungen des Undo-Modells abdeckt. Bei der Wahl eines *linearen baumbasierten Undo-Modells* muss er beispielsweise die Benutzerabfrage, welcher Ast traversiert wird, in die Anwendung mit einbeziehen.

Manche Operationen können nicht rückgängig gemacht werden (vgl. 2.8.5). Welche das sind, weiß der Entwickler und so kann er auch vor Ausführung der Operation eine entsprechende Meldung an den Benutzer ausgeben lassen. Ebenso ist sich der Entwickler darüber im Klaren, welche Operationen in die Befehlsgeschichte aufgenommen werden müssen. Das realisierte MITK-Undo-Konzept bietet ihm die Möglichkeit, alle Informationen zum Undo und Redo zu speichern.

Eine wesentliche Anforderung besagt, dass alle nötigen Elemente jeder Operation ihre Ausführung überdauern müssen (vgl. 2.8.4). Die Basis-Idee der Realisation dieser Anforderung und zugleich auch anderer Anforderungen besteht in der *Kommunikation über Objekte*. Dadurch lässt sich Folgendes realisieren:

- Operationen sind gekapselt und können auch von anderen Objekten aus ausgeführt werden.
- Undo- und Redo-Information können gruppiert werden.
- Diese Gruppe kann in einer Befehlsgeschichte gespeichert werden.

Im Folgenden gehen wir nun detaillierter auf diese und andere Punkte ein.

3.3.1 Kapselung der Operationen

Für die Realisation des Undo-Konzepts müssen wir alle Anweisungen der Operationen, die undo-fähig sein sollen, kapseln. In den Methoden der Interaktions-Klassen muss die Ausführung der Operationen über einen Methodenaufruf realisiert und die Parameter, die für die Operation benötigt werden, mit übergeben werden.

Erläutern wir es an einem Beispiel: Das Objekt `SetOfPointsInteractor` stellt eine Zustandsmaschine mit definiertem Verhalten dar. Die Durchführung eines Zustandswechsels hat eine bestimmte Aktion zur Folge. Die in dem Übergang gespeicherte Aktions-Identifikationsnummer `SideEffektId` wird der Methode `ExecuteSideEffect(..)` übergeben und spezifiziert die auszuführende Aktion. Hier wird die Kapselung vorgenommen: In der Methode werden keine Daten, wie z.B. Punktkoordinaten, verändert, sondern es wird eine weitere Methode `ExecuteOperation(..)` des Daten-Objekts aufgerufen, der die nötigen Parameter mit übergeben werden. In `ExecuteOperation(..)` werden daraufhin die Daten verändert.

Durch diese Kapselung erreichen wir, dass wir eine undo-fähige Operation sowohl zu jeder Zeit als auch mit unterschiedlichen Parametern als auch von jedem Objekt aus ausführen lassen können.

Ein Objekt, das Operationen ausführt und somit die Methode `ExecuteOperation(..)` beinhaltet, ist von dem Interface `OperationActor` abgeleitet und überschreibt die abstrakte Methode. Ein Beispiel ist das Objekt `StateMachine`, in dem die Möglichkeit, in einen vorigen Zustand zu gelangen, über die Methode `ExecuteOperation(..)` gewährleistet ist.

3.3.2 Operations-Klassen

Die Kommunikation über Objekte wird durch die Gruppierung der Elemente einer Operation in einem *Operations-Objekt* bewerkstelligt. Die Definition einer einzigen Klasse ist

nicht von Vorteil, da die Klasse bei der Vielzahl von unterschiedlichen Operationen die Anzahl aller Elemente beinhalten müsste. Deshalb müssen unterschiedliche Operationstypen deklariert werden, dessen Aufbau sich nach den Operations-Arten richtet.

Die Basis-Klasse aller Operationen stellt die Klasse `Operation` dar, welche nur eine Membervariable des Typs `string` bzw. `char[]` beinhaltet (vgl. Abb. 3.21). In dieser Variable ist die Information über die Art der Operation (z.B. Löschen, Verschieben) festgehalten.

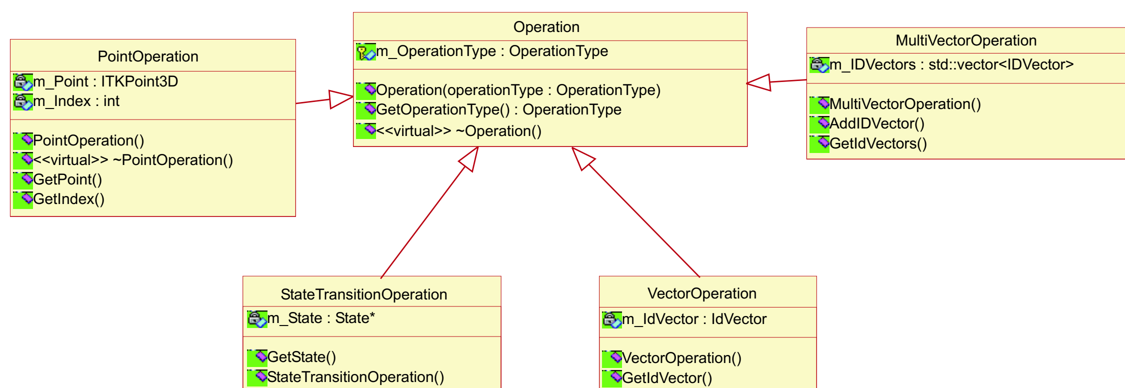


Abbildung 3.21: Klassen-Diagramm der Operations-Klassen

Von der Basis-Klasse abgeleitet werden Klassen aller Operations-Art. Beispiele derer sind die Klassen `PointOperation`, `StateTransitionOperation`, `VectorOperation` und `MultiVectorOperation`. Betrachten wir uns die Klasse `PointOperation` als ein Beispiel für eine Operations-Art. Es enthält einen dreidimensionalen Punkt und einen Index, mit dem eine Position eines Elements in einer Liste angegeben werden kann. Die Membervariablen werden über den Konstruktor beschrieben und ihr Inhalt über entsprechende Get-Methoden ausgelesen. Ein Objekt dieser Klasse dient zur Übermittlung der Informationen, die für die Veränderung bzw. Erstellung eines Punktes nötig sind.

Das Objekt des Typs `StateTransitionOperation` beinhaltet eine Referenz eines Zustands. Durch dieses Objekt wird in dem zuvor beschriebenen Interaktionskonzept der Zeiger des resultierenden Zustands einer Methode, die den Zustandswechsel vornimmt, übergeben.

Die übrigen Klassen sind auf andere Operations-Arten spezialisiert.

3.3.3 Speicherung in der Befehlsgeschichte

Unabhängig von dem Undo-Modell müssen wir bei der *Strategie der inversen Operation* das in 3.3.2 beschriebene Objekt `Operation` in einer Befehlsgeschichte speichern. So überdauert die Information der Operation ihre Ausführung. Hierzu müssen wir eine klare Zuordnung der Inversen zu ihrer Operation bzw. der Undo- zu der Redo-Operation vornehmen.

Das Objekt `OperationEvent` übernimmt diese Zuordnung (vgl. Abb. 3.22). Hier werden

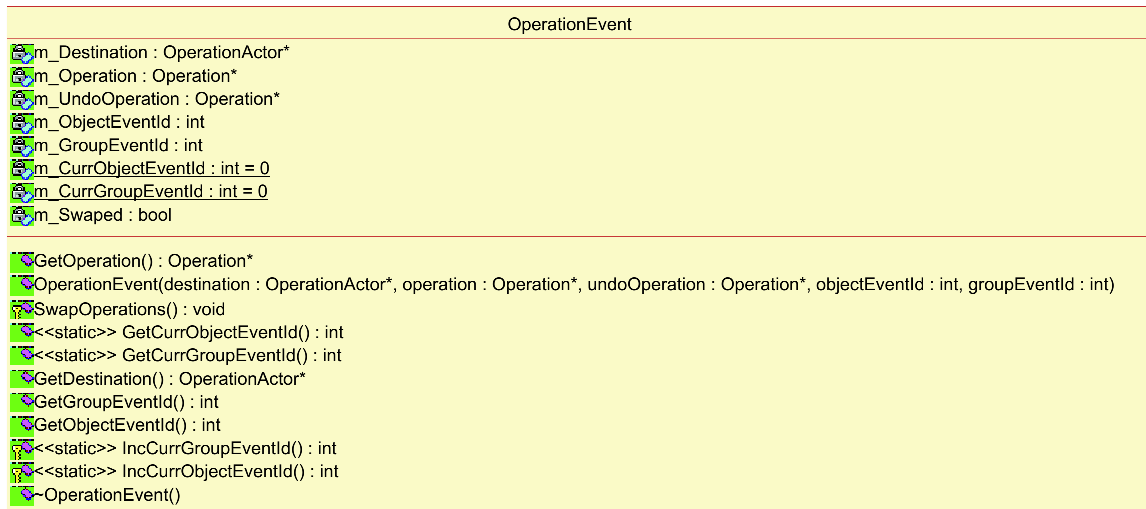


Abbildung 3.22: Klassen-Diagramm OperationEvent

die Referenzen der Undo- und der Redo-Objekte in den Membervariablen `m_UndoOperation` und `m_Operation` gespeichert. Um einen Meta-Befehl Undo auszuführen, wird die Information benötigt, an welches Datum die Operation geschickt werden soll. Die Membervariable `m_Destination` enthält die Referenz dieses Datums.

Eine Operation sollte unbeabsichtigt nicht zweimal rückgängig gemacht werden. Daher muss gewährleistet werden, dass Undo und Redo immer in Folge aufgerufen werden. Die Methode `SwapOperations(..)` ermöglicht dies. Wird nach jedem Undo- und Redo-Befehl die Methode ausgeführt, so kann immer die Referenz `m_Operation` des Objekts genutzt werden, um an die richtige Operation zu gelangen. Durch den Methodenaufruf `SwapOperations(..)` werden die Zeiger `m_Operation` und `m_UndoOperation` vertauscht. Dabei wird das Vertauschen nicht nach jedem Zugriff auf `m_Operation` vorgenommen, denn sonst wäre es nur umständlich möglich, auf die Informationen der Operation zuzugreifen, ohne ein Undo auszuführen. Die boolsche Membervariable `m_Swaped` wird nach einem Vertauschen invertiert, so dass hierüber auch der Inhalt der Referenzen `m_Operation` und `m_UndoOperation` reproduzierbar ist.

Durch die Problematik, zu jeder Operation eine inverse zu finden, wird das Speichern der Operationen in der Befehlsgeschichte nur angeboten, d.h. das Interaktionskonzept ist auch ohne Undo-Konzept funktionstüchtig.

Ein Beispiel (vgl. Abb. 3.23): Das Objekt `stateMachine` erstellt zwei Objekte: `doOperation` und `undoOperation`. Das Objekt `doOperation` wird über einen Methodenaufruf an das Daten-Objekt `baseData` übergeben. Hier werden die Informationen aus `doOperation` für die Änderung der Daten (z.B. Punktkoordinaten) verwendet. Daraufhin werden `doOperation` und `undoOperation` in einem Objekt `operationEvent` zusammenge-

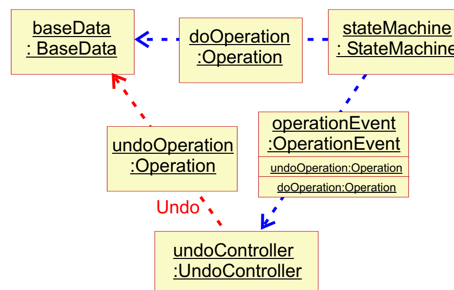


Abbildung 3.23: Objekt-Diagramm der Kommunikation zwischen Interaktions- und Daten-Objekten; *roter Pfeil*: Referenz; *blau gestrichelt*: Nachricht

fasst, an das Objekt `undoController` gesendet und dort in die Befehlsgeschichte eingetragen. Bei einem Undo-Befehl wird das Objekt `operationEvent` aus der Liste geholt und das darin enthaltene Objekt `undoOperation` an das Daten-Objekt `baseData` geschickt. Die Daten werden daraufhin so verändert, dass der vorherige Zustand wiederhergestellt ist.

In einem anderen Ansatz könnten die Informationen zum Ändern der Daten von dem Objekt `stateMachine` über `undoController` an das Objekt `baseData` gesendet werden. So wäre das Interaktionskonzept ohne das Undo-Konzept allerdings nicht möglich.

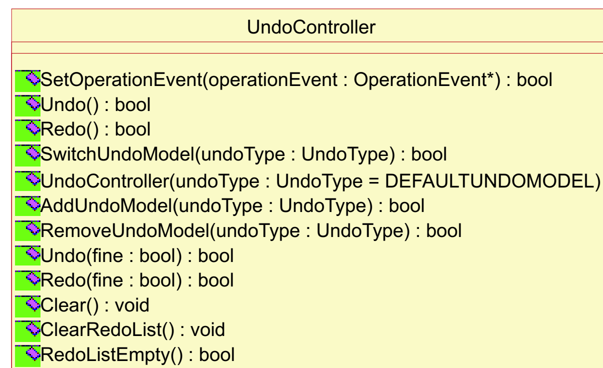


Abbildung 3.24: Klassen-Diagramm UndoController

Das Objekt der Klasse `UndoController` (vgl. Abb. 3.24) übernimmt die Verwaltung der Undo-Modelle. In eine interne Liste können Modelle hinzugefügt und gelöscht werden. Die Befehlsgeschichte des ausgewählten Modells kann insgesamt (Undo und Redo) oder zur Unterstützung von Undo-Modellen mit stabiler Ausführungseigenschaft (vgl. 2.8.3) nur auf die Redo-Liste bezogen gelöscht werden. Durch die Methode `SwitchUndoModel` kann zu einem bestimmten Modell gewechselt werden. Im laufenden Betrieb muss jedoch das Modell auf Konsistenz geprüft werden bzw. die Befehlsgeschichte gespeichert oder sogar gelöscht werden. In der Methode `SetOperationEvent(..)` wird das als Parameter übergebene Objekt `operationEvent` der Befehlsgeschichte des aktuellen Undo-Modells hinzugefügt.

Der Entwickler entscheidet, ob eine Operation undo-fähig ist. Ist dies nicht der Fall, so muss nur ein Operations-Objekt erstellt werden, welches dann an das Datum geschickt wird. Die restlichen Objekte, `undoOperation` und `operationEvent`, werden hier nicht benötigt. Davon soll jedoch nur in seltenen Fällen Gebrauch gemacht werden. Eine feingranuläre Aufschlüsselung der Operationen unterstützt das Finden von inversen Operationen.

3.3.4 Angebot von Undo-Modellen

Dem Entwickler stehen im MITK mehrere Undo-Modelle zur Verfügung. Er kann ein bestimmtes Modell oder auch eine Anzahl von Modellen wählen, um während der Laufzeit dem Benutzer die Möglichkeit der Wahl zu geben. Selbst wenn das präferierte Modell nicht angeboten wird, so ist es dem Entwickler möglich, dieses unterstützt zu implementieren. Hierfür steht eine Klasse `UndoModel`, von dem die Klasse des neuen Modells abgeleitet wird, zur Verfügung. Dieses Interface beinhaltet abstrakte Methoden, wie `Undo` oder `Redo`, die implementiert werden müssen. Dadurch kann das neue Modell problemlos in die MITK-Umgebung aufgenommen werden.

3.3.5 Eingeschränktes lineares Undo-Modell

Als erstes Undo-Modell wurde das *eingeschränkte lineare Undo-Modell* implementiert. Es zeichnet sich, wie in 2.8.3 beschrieben, durch ein transparentes Verhalten aus.

Die Befehlsgeschichte wird durch zwei Listen realisiert: Undo- und Redo-Liste. Die Objekte des Typs `OperationEvent` werden der Reihe nach in der Undo-Liste gespeichert. Jede Liste ist durch einen Kellerspeicher (stack) repräsentiert, in dem nur an einem Ende hinzugefügt (push) und herausgenommen (pop) werden kann.

Wird die Methode `Undo(..)` aufgerufen, so wird das zuletzt hinzugefügte Objekt aus der Liste genommen (pop), die darin enthaltene Undo-Operation (Inverse) dem Daten-Objekt übergeben, die Methode `SwapOperations(..)` aufgerufen und das Objekt `OperationEvent` in die Redo-Liste geschrieben (push). Bei einem Redo wird ähnlich verfahren, nur wird das Objekt aus der Redo-Liste geholt, die Operation verschickt, `SwapOperations(..)` aufgerufen und in die Undo-Liste geschrieben.

Plausibilitätskontrollen der Operationen müssen nicht durchgeführt werden, da das Modell eine stabile Ausführungseigenschaft besitzt (vgl 2.8.3). Soll ein neues Objekt in die Undo-Liste aufgenommen werden, so muss überprüft werden, ob die Redo-Liste leer ist.

3.3.6 Granularität des Undos

Ein vom Benutzer ausgeführter Befehl kann mehrere Operationen zur Folge haben. Durch die feingranuläre Auflösung der Operationen im MITK-Interaktionskonzept wird dies noch verstärkt. Ein Beispiel: Der Benutzer betätigt die Taste „Del“ (Entf) und will dadurch einen

selektierten Punkt löschen. Das Ereignis „Del_wurde_gedrückt“ wird in den Interaktions-Objekten weiterverarbeitet und führt zu mehreren Operationen. Die erste Operation, die ausgeführt wird, kann einen Zustandswechsel der globalen Zustandsmaschine bewirken. Hieran kann sich eine Aktion-Operation anschließen, in der der Punkt gelöscht wird. Andere Maschinen können das Event benötigen, um weitere Operationen zu erstellen. So entstehen mehrere Operationen aus einem vom Benutzer ausgelösten Befehl.

Im Falle eines Undos sollen alle zu einem Befehl gehörigen Operationen rückgängig gemacht werden. Besonders die Operation „Zustandswechsel“ soll immer mit ihrer zugehörigen Aktion-Operation widerrufen werden. Deswegen muss in dem Konzept auch eine Möglichkeit geschaffen werden, die zueinander gehörenden Operationen zu identifizieren.

Dies wurde dadurch bewerkstelligt, dass in dem Objekt `OperationEvent` (vgl. 3.3.3) eine Nummer `m_ObjectEventId`, die eine Gruppe identifiziert, gespeichert wird. Diese Nummer muss eindeutig sein. Daraus folgt, dass die Nummer in einem zentralen Objekt gehalten werden muss. Damit die Nummer von allen Objekten aus erreicht werden kann, wird eine Methode `GetCurrObjectEventId` deklariert, in der die statische aktuelle Nummer `m_CurrObjectEventId` angefordert werden kann. Diese Methode soll in einer Klasse deklariert sein, die von allen Objekten, in denen undo-fähige Operationen erstellt werden, eingebunden wird. Die Klasse `OperationEvent` ist für dieses Problem die nahe liegende Lösung. Vor dem Aufbau eines Objekts `OperationEvent` wird über die statische Methode `GetCurrObjectEventId(..)` die aktuelle Objekt-Identifikationsnummer erfragt und in dem Konstruktor-Aufruf des neu zu erstellenden Objekts als Parameter übergeben. Jedes Undo-Modell macht alle Operationen rückgängig, die die gleiche Identifikationsnummer wie das zuerst aus der Undo- bzw. Redo-Liste geholte `OperationEvent`-Objekt besitzen. Somit ist ein vom Benutzer getätigter Befehl rückgängig gemacht.

Die statische Variable `m_ObjectEventId` muss jedoch auch von einem Objekt inkrementiert werden, um die unterschiedlichen Befehle zu unterscheiden. Ein Befehl, wie etwa das Betätigen einer Taste, löst ein Ereignis aus. Dieses wird von dem Objekt `EventMapper` empfangen, welches dann das Ereignis an die weiterverarbeitenden Objekte übermittelt. Das Objekt `EventMapper` ist somit das erste MITK-Objekt, das bei einem Befehl benachrichtigt wird. Hier muss also auch die Variable `m_ObjectEventId` um eins angehoben werden. Durch den Aufruf der statischen Methode `IncCurrObjectEventId(..)` des Objekts `OperationEvent` wird dies ausgeführt.

Eine Erweiterung des Konzepts sieht vor, dass nicht nur die Operationen eines Befehls, sondern auch die einer Gruppe von Befehlen widerrufen werden können. Eine Anwendung dieser Erweiterung ist das Undo aller für den Aufbau eines Polygons ausgeführten Operationen durch einen einzigen Meta-Befehl. Dabei ist es dem Benutzer möglich zu entscheiden, ob er einen Befehl oder eine Gruppe von Befehlen rückgängig machen will.

Die Erweiterung sieht lediglich das Speichern einer zweiten Variable `m_GroupEventId` in dem Objekt `OperationEvent` vor. Die entsprechenden statischen Methoden und die statische Variable `m_CurrGroupEventId` werden wie die zuvor beschriebenen bereitgestellt.

In einer rudimentären Test-Applikation wurde hiermit ein zwar weniger funktioneller, dafür aber anschaulicher Einsatz implementiert. Ohne Inkrementieren der Gruppen-Identifikationsnummer wurden durch einen speziellen Befehl alle bisher ausgeführten Operationen rückgängig gemacht. Die Ansicht der Applikation entsprach einem zu schnell rückwärts abgespielten Film, der den Ablauf des Programms seit Start zeigte.

3.3.7 Gesamtübersicht des Undo-Konzepts

Die folgende Abbildung 3.25 bietet den Gesamtüberblick und eine Zusammenfassung aller wichtigen Undo-Klassen. Die einzelnen Klassen sind nummeriert und werden im Folgenden kurz beschrieben.

1 Operation stellt die Basis-Klasse aller Operationen dar; enthält eine Angabe über die Art der Operation (Löschen, Selektieren etc.)

2 PointOperation eine Operation, durch die ein Punkt verändert wird; enthält einen Index für die Position eines Punktes in einer Liste

3 StateTransitionOperation Operation, durch die ein Zustandswechsel einer Zustandsmaschine durchgeführt wird

4 DisplayCoordinateOperation enthält mehrere Koordinaten, die für eine multiplanare Rekonstruktion (MPR) benötigt werden

5 VectorOperation enthält einen Vektor, der für eine Operation benötigt wird

6 MultiVectorOperation enthält mehrere Vektoren, die für eine Operation benötigt werden

7 OperationEvent beinhaltet die Undo- und die Redo-Operation; enthält Objekt- und Gruppen-Identifikationsnummer; wechselt Undo- und Redo-Zeiger

8 UndoController verwaltet alle Undo-Modelle

9 Undo-Model Interface aller Undo-Modelle

10 LimitedLinearUndo Realisation des eingeschränkten linearen Undo-Modells

11 TreeUndo noch nicht realisierte Klasse des baumbasierten Undo-Modells als Beispiel für Erweiterungen

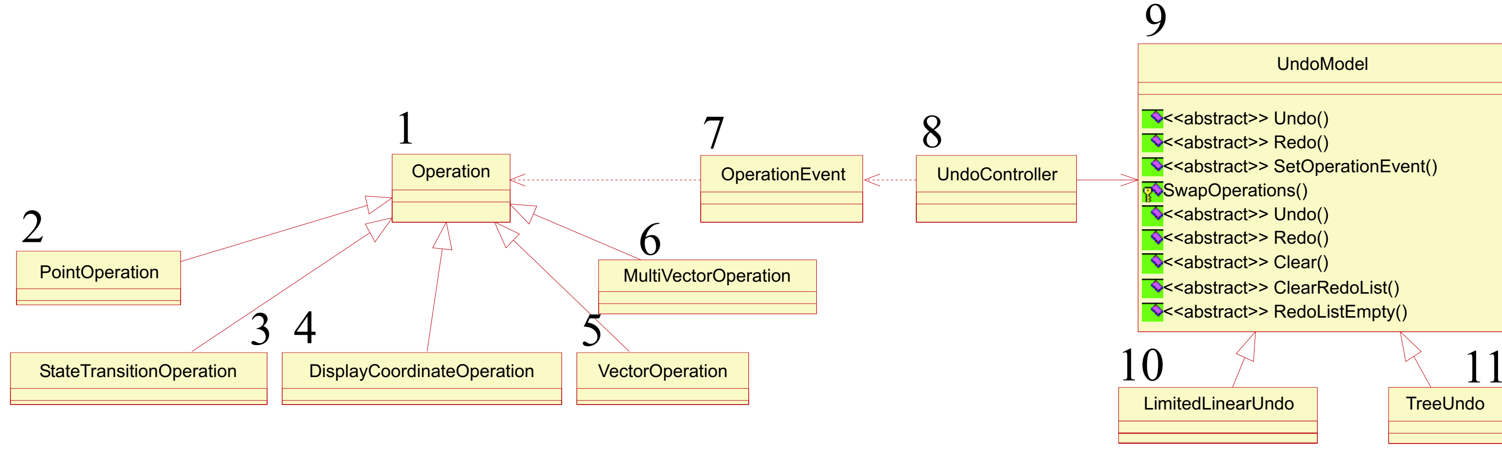


Abbildung 3.25: Übersicht der wichtigsten Undo-Klassen

Kapitel 4

Implementierung

Im Folgenden werden einige Auszüge des Quellcodes wichtiger Klassen angesprochen. Die Beispiele sind objektorientiert in C++ geschrieben [KDDS96] [JCJv92]. Kommentare, die im Quelltext diesen dokumentieren, sind der besseren Übersicht wegen überwiegend entfernt. Ebenso sind Sicherheitsabfragen sowie Überprüfungen gegen `NULL` entfernt, wenn es nicht für die Aufgabenerfüllung der Methode wichtig ist. Ausgaben an den Benutzer sind schematisch dargestellt und in ihrer Aussagekraft gekürzt. Sie sollen in einer späteren Phase des Projekts durch Nummern, die einen bestimmten Text identifizieren, ersetzt und ausgelagert werden.

Der Quellcode ist durchgehend dokumentiert. Die gesamte Dokumentation soll in Zukunft der Öffentlichkeit zur Verfügung gestellt werden.

4.1 Konventionen im MITK

Für die bessere Lesbarkeit dieses Kapitels werden nun einige Konventionen, die von MITK gefordert werden, aufgelistet.

Die Klassen des MITKs stehen unter dem *NameSpace* „mitk::“. Membervariablen werden mit dem Präfix `m_` und einer eindeutigen Bezeichnung gekennzeichnet. Der erste Buchstabe nach dem Präfix wird groß geschrieben. Setzt sich der Name aus mehreren Begriffen zusammen, so werden diese möglichst verständlich abgekürzt und jeder Anfangsbuchstabe groß geschrieben. Hierdurch ist eine klare Trennung zwischen Membervariablen und lokalen Variablen möglich. Beispiele: `m_Id`, `m_Event`, `m_NextStateId`

Parameter werden klein geschrieben. Zur besseren Kennzeichnung werden Anfangsbuchstaben neuer Wörter groß geschrieben. Beispiele: `id`, `event`, `nextStateId`

Klassennamen, Methodennamen sowie Typnamen werden groß geschrieben. Beispiele: `GetId`, `IsEvent`, `SetNextStateId` und `EventList`, `TransitionMap`, `StateListIter`

Konstanten, die für die Implementierung benötigt werden, sind in einer separaten Datei definiert und werden zur Kennzeichnung in Großbuchstaben geschrieben. Konstanten der Operationen sind mit Präfix „Op“, der Aktionen mit „Se“ gekennzeichnet. Beispiele: `OpDELETE`, `SeCHECKX`

4.2 Implementierung des Interaktions-Modells

Das generische Interaktionskonzept umfasst viele Klassen. Die Ideen, die hinter den Klassen stecken, sind im vorherigen Kapitel vorgestellt worden. Aufbauend darauf werden im Folgenden Auszüge des Quellcodes der wichtigsten Klassen vorgestellt.

4.2.1 Zustandswechsel

Die Methode `HandleEvent(..)` der Klasse `StateMachine` übernimmt den Wechsel des aktuellen Zustands einer Zustandsmaschine (vgl. Abb. 4.1).

```
bool mitk::StateMachine::HandleEvent(StateEvent const* stateEvent,
  int objectEventId, int groupEventId)
{
 const Transition *tempTransition =
   m_CurrentState->GetTransition(stateEvent->GetId());

 if (tempTransition == NULL)
    return false;

 State *tempNextState = tempTransition->GetNextState();
 int tempSideEffectId = tempTransition->GetSideEffectId();

 if ((m_CurrentState->GetId()!=tempNextState->GetId())
   &&(m_UndoEnabled))
 {
  StateTransitionOperation* doOp =
    new StateTransitionOperation(OpSTATECHANGE, tempNextState);
  StateTransitionOperation* undoOp =
    new StateTransitionOperation(OpSTATECHANGE, m_CurrentState);
  OperationEvent *operationEvent =
    new OperationEvent(((mitk::OperationActor*)(this)),
      doOp,undoOp,objectEventId,groupEventId);
  m_UndoController->SetOperationEvent(operationEvent);
 }

 m_CurrentState = tempNextState; //this->ExecuteSideEffect(doOp);
 bool ok = ExecuteSideEffect(tempSideEffectId, stateEvent,
   objectEventId, groupEventId);

 if (!ok && m_UndoEnabled)
 {
  ok = m_UndoController->Undo(true);//fine undo!
 }
 else if (!ok && !m_UndoEnabled)
 {
  std::cout<<"Error!"<<std::endl;
 }
 return ok;
}
```

Abbildung 4.1: Methode HandleEvent der Klasse StateMachine

Der Methode übergeben werden das Ereignis `stateEvent` und die Nummern `objectEventId` sowie `groupEventId`. In `stateEvent` sind die Informationen über das Ereignis und die interne Ereignis-Identifikationsnummer des MITKs enthalten. Zunächst wird über die Identifikationsnummer der entsprechende Übergang gesucht. Wird kein Übergang

gefunden, so wird das Ereignis nicht akzeptiert und die Methode mit dem Rückgabewert `false` beendet. Wird ein Übergang zu dem Ereignis gefunden, so wird die im Übergang enthaltene Referenz auf den nächsten Zustand einer lokalen Referenz zugewiesen. Über die Methode `GetSideEffectId(..)` wird die Aktions-Identifikationsnummer des Übergangs in eine weitere lokale Variable geschrieben.

Damit nicht unnötig Instanzen von Objekten erstellt werden, die in der Befehlsgeschichte gespeichert werden müssen, wird im Anschluss überprüft, ob es sich bei dem Übergang um einen Übergang in denselben Zustand handelt. Ist dies nicht der Fall und ist die für eine Aktivierung des Undos bereitgestellte Variable `m_UndoEnabled` auf `true` gesetzt, so werden zwei Operations-Objekte aufgebaut, die gruppiert in einem `OperationEvent`-Objekt für ein späteres Undo nötig sind. Das Objekt `operationEvent` wird dem `UndoController` übergeben und in ihm in die Befehlsgeschichte aufgenommen.

In anderen Interaktions-Objekten wird das erstellte Operations-Objekt an ein Daten-Objekt gesendet. Hier ist das Objekt des Typs `StateMachine` beides, Interactor und Datum. Für das Undo muss die Methode `ExecuteOperation(..)` bereitgestellt sein, denn nur über die Methode kann der Zeiger des aktuellen Zustands versetzt werden. In der Methode `HandleEvent(..)` haben wir jedoch Zugriff auf die Membervariable `m_CurrentState` und können uns so einen Methodenaufruf sparen und direkt den Zeiger versetzen (vgl. Abb. 4.1, Zeile 15).

Nach der Ausführung der Operation wird die Aktion in der objekteigenen Methode `ExecuteSideEffect(..)` ausgeführt. Der boolsche Rückgabewert gibt Aufschluss darüber, ob die Aktion erfolgreich ausgeführt wurde. Ist dies nicht der Fall, so soll in den vorherigen Zustand zurückgekehrt werden. Ein Undo mit dem Parameter `true` als Signal für das Undo aller Operationen mit der gleichen `ObjectEventId` führt dies durch. Die Abfolge erscheint auf den ersten Blick umständlich. Zuerst führen wir die Operation durch, um sie bei einer nicht ausgeführten Aktion wieder rückgängig zu machen. Jedoch kann die Aktion auch vom aktuellen Zustand abhängig sein und so muss der Zustandswechsel zuerst vorgenommen werden. Zudem wird unterschieden, ob die Undo-Funktionalität in dieser Zustandsmaschine aktiviert ist oder nicht. Standardmäßig ist sie es, jedoch kann sie zu Testzwecken oder in Ausnahmesituationen deaktiviert werden. Ist sie nicht aktiviert, so kann auch kein Undo ausgeführt werden und es muss eine Meldung an den Benutzer ausgegeben werden.

Beendet wird die Methode `HandleEvent(..)` mit der Übergabe des empfangenen boolschen Rückgabewerts der Methode `ExecuteSideEffect(..)`.

Der in 3.2.9 beschriebene Null-Übergang ist in der Klasse `State` realisiert und der soeben beschriebenen Methode nicht anzusehen. In der Methode `GetTransition(..)` der Klasse `State` wird zunächst geprüft, ob ein Übergang mit der übergebenen Identifikationsnummer existiert. Ist dies nicht der Fall, so wird in einem zweiten Versuch nach einem Übergang mit $ID = 0$ gesucht. Erst wenn beide Versuche ohne Erfolg waren, wird `NULL` von der Methode

`GetTransition(..)` zurückgegeben.

4.2.2 Generierung der Verhaltensmuster

Für die Entwicklung der grafischen Benutzerschnittstelle wird Trolltechs *QT* [Tro03a] (vgl. 1.2) verwendet. QT bietet Objekte (z.B. `QXmlDefaultHandler`) für die Analyse von XML-Dateien an, so dass wir die Möglichkeit haben, durch geringen Programmieraufwand ein Objekt zu erstellen, welches zur Laufzeit die XML-Datei untersucht.

Die Klasse `StateMachineFactory` wurde bereits in 3.2.3 beschrieben. Sie wird von der Klasse `QXmlDefaultHandler` abgeleitet. Die überschreibende Methode `LoadBehavior(..)` bereitet die Analyse der XML-Datei vor. In der Methode `startElement(..)`[1] werden die Objekte der Zustände und der Übergänge aufgebaut. Dabei dienen die Attribute der XML-Informationen als Membervariablen der Objekte.

Um den Aufbau der Zustandsmaschine näher kennen zu lernen, gehen wir nun auf die Methode `ConnectStates(..)` ein (vgl. Abb. 4.2).

Im Parameter `states` wird eine Liste aller Zustände und Übergänge einer Zustandsmaschine übergeben. Diese sind untereinander noch nicht referenziert.

Zuerst wird das übergebene Konstrukt einem Test unterzogen, der überprüft, ob alle Zustände über die Übergänge erreicht werden können (vgl. 4.2.3). Je nach Ergebnis wird eine Mitteilung an den Benutzer geschickt.

Im Anschluss wird in allen Übergängen der übergebenen Zustände die Referenz des nächsten Zustands gesetzt. In den Übergängen steht bisher nur die Identifikationsnummer des nächsten Zustands. Jedoch soll das Konstrukt durch Zeiger traversierbar sein. Die Referenz kann nur innerhalb jedes Objekts `State` gesetzt werden. Hierfür steht eine Methode `ConnectStates(..)` zur Verfügung (vgl. Abb. 4.2, Zeile 27). Eine Liste `allStates` aller aufgebauten Zustände mit der zugehörigen ID wird der Methode übergeben. In der Methode wird in allen Übergängen eines Zustands die Referenz des nächsten Zustands gesetzt. Somit besitzt ein Zustand eine Liste aller Übergänge, von denen jeder eine Referenz auf den resultierenden Zustand enthält.

4.2.3 Test der Zustandsmaschine

Die Möglichkeit des Tests wurde in die Planung mit einbezogen. Ein Test, der die Definition der Zustandsmaschine überprüft, ist bereits integriert (vgl. Abb. 4.3), weitere können implementiert werden.

Die Methode `parse(..)` der Klasse `StateMachineFactory` unterzieht die erstellten Objekte `State` und `Transition` einem einfachen Test. Dieser Test geht die Zustände der Zustandsmaschine rekursiv nach Identifikationsnummer durch.

[1] Methoden-Namen fangen bei QT klein an.

```
1  bool mitk::StateMachineFactory::ConnectStates(mitk::State::StateMap *states)
2  {
3   if(states->size()>1)
4   {
5    HistorySet *history = new HistorySet;
6    mitk::State::StateMapIter firstState = states->begin();
7    bool ok = parse(states,firstState,history);
8    delete history;
9    if((states->size() != history->size()) || !ok)
10   {
11    std::cout<<"Error!"<<endl;
12   }
13  }
14
15  for (mitk::State::StateMapIter tempState = states->begin();
16   tempState!=states->end(); tempState++)
17  {
18   bool tempbool = ((tempState->second)->ConnectTransitions(states));
19   if(tempbool = false)
20   {
21    return false;
22    std::cout<<"Error!"<<endl;
23   }
24  }
25  return true;
26 }
27
28 bool mitk::State::ConnectTransitions(StateMap *allStates)
29 {
30  for (TransMapIter i = m_Transitions.begin(); i != m_Transitions.end(); i++)
31  {
32    StateMapIter sIter = allStates->find(((*i).second).GetNextStateId());
33   if(sIter != allStates->end())
34    ((*i).second).setNextState((*sIter).second);
35     else
36      return false;
37   }
38  return true;
39 }
```

Abbildung 4.2: Methode ConnectStates der Klasse StateMachineFactory und Methode ConnectTransitions der Klasse State

```
bool mitk::StateMachineFactory::parse(mitk::State::StateMap *states,
  mitk::State::StateMapIter thisState, HistorySet *history)
{
 history->insert((thisState->second)->GetId());
 std::set<int> nextStatesSet = (thisState->second)->GetAllNextStates();
 std::set<int>::iterator position =
   nextStatesSet.find((thisState->second)->GetId());

 if (position != nextStatesSet.end())
 {
  nextStatesSet.erase(position);
 }

 if ( nextStatesSet.empty() )
 {
  std::cout<<"Error!"<<endl;
  return true;
 }

 bool ok;
 for (std::set<int>::iterator i = nextStatesSet.begin();
  i != nextStatesSet.end(); i++)
 {
  if ( history->find(*i) == history->end() )
  {
   mitk::State::StateMapIter nextState = states->find(*i);
   ok = parse( states, nextState, history);
  }
 }
 return ok;
}
```

Abbildung 4.3: Testen einer Zustandsmaschine; die Methode parse der Interaktions-Klasse StateMachineFactory

Übergeben werden der Methode die Liste `states` aller aufgebauten `State`-Objekte, ein *Iterator* `thisState`, der auf einen Eintrag dieser Liste zeigt, und eine Liste `history`, die die Zustands-Identifikationsnummern der bereits überprüften Zustände beinhaltet. Zunächst wird die Identifikationsnummer des Zustands, der getestet wird, in die Liste `history` geschrieben. In der Liste werden alle bisher durchlaufenden Zustands-Nummern gehalten, um zu verhindern, dass ein Zustand zweimal überprüft wird. Hierauf wird lokal eine Liste `nextStatesSet` aller Zustands-Identifikationsnummern, die über die Übergänge von diesem Zustand aus erreicht werden, erstellt. Als nächstes wird überprüft, ob ein Übergang in denselben Zustand existiert. Falls ein solcher Übergang gefunden wird, so wird er aus der lokalen Variable `nextStatesSet` entfernt. Hierüber können wir einen *toten Zustand* (vgl. 2.7.5) ausfindig machen, denn die nächste Bedingungsanweisung überprüft, ob die Liste `nextStatesSet` nach dem evtl. Löschen leer ist. Ist sie es, so führt kein Übergang aus diesem Zustand heraus. Diese Konstruktion kann natürlich auch im Falle eines finalen Zustands gewollt sein, jedoch ist so ein Aufbau für ein interaktives System eher ungewöhnlich. Der Benutzer wird deshalb über den Fund benachrichtigt. Die folgende rekursive Überprüfung kann übergangen werden, da kein Eintrag mehr in der Liste aller nächsten Zustände existiert. Deshalb wird der Test in dieser Ebene der Rekursion beendet.

Zuletzt wird der rekursive Aufruf über alle Zustände, die über die Übergänge des momentanen Zustands erreicht werden, jedoch noch nicht überprüft wurden, durchgeführt.

Dieser Test findet *tote* sowie *magische Zustände* (vgl. 2.7.5), denn nach Beendigung des Tests wird die Größe der Listen `history` und `states` verglichen (vgl. Abb. 4.2, Zeile 7). Ist die Größe beider Listen unterschiedlich, so wurde mindestens ein Zustand nicht erreicht. Der Test wird allein über die Identifikationsnummer der Zustände und nicht über ihre Referenz durchgeführt. Nach dem Test kann entschieden werden, ob die Übergänge mit den resultierenden Zuständen über Referenz verbunden oder alle Objekte der Zustandsmaschine gelöscht werden.

4.2.4 Wiederverwendung der Interaktionsmuster

Die Ereignisse, die vom Benutzer ausgeführt werden, sind in einer interaktiven Anwendung nicht vorhersehbar. Deswegen empfiehlt es sich, die Zustandsmaschinen während der Laufzeit zu erzeugen [GHJV96, S. 403]. Dadurch wird vermieden, dass Objekte, die nicht gebraucht werden, unnötig Ressource rauben.

Auch wenn der Aufbau zur Laufzeit schon Speicherplatz spart, müssen wir die Ressourcen, die die vielen Maschinen benötigen, weiter reduzieren. Ein Ansatzpunkt für eine Reduzierung wird durch die Vielzahl gleicher Zustandsmaschinen geboten. Wie wir schon in 3.2.5 besprochen haben, können sich Maschinen zwar nicht die Logiken, aber die Verhaltensmuster teilen. So können wir den größten Teil einer Zustandsmaschine wiederverwenden und damit, wie benötigt, Ressourcen sparen.

```
1  mitk::State* mitk::StateMachineFactory::GetStartState(std::string type)
2  {
3   StartStateMapIter tempState = m_StartStates.find(type);
4   if( tempState != m_StartStates.end() )
5    return (tempState)->second;
6   else
7    return NULL;
8  }
9
10 mitk::StateMachine::StateMachine(std::string type) : m_Type(type)
11 {
12  m_CurrentState = mitk::StateMachineFactory::GetStartState(type);
13
14  m_UndoEnabled = true;
15 }
```

Abbildung 4.4: Wiederverwendung der Interaktionsmuster

Die Klasse `StateMachineFactory` bietet alle initialen Zustände der aufgebauten Verhaltensmuster (vgl. Abb. 4.4) an. Durch den Aufruf der Methode `GetStartState(..)` und der Übergabe des Parameters `type`, der das Muster identifiziert, wird die Referenz auf den Start-Zustand des Musters gesucht und falls gefunden zurückgegeben.

Bei jedem neuen Aufbau einer Zustandsmaschine wird diese Methode im Konstruktor `StateMachine(..)` aufgerufen (vgl. Abb. 4.4, Zeile 13) und die zurückgelieferte Referenz in der globalen Variable `m_CurrentState` gespeichert. Somit ist nicht zu erkennen, dass andere Zustandsmaschinen auf das gleiche Konstrukt aus `State`- und `Transition`-Objekten zugreifen, aber die Ressourcen sind minimiert.

4.2.5 Beispiel für die Modellierung der Interaktion

Das Zusammenwirken der vielen Objekte, besonders bei erweiterten Zustandsmaschinen-Hierarchien, ist komplex. Im folgenden Beispiel wird der Ablauf der Interaktion, in dem der Benutzer eine maximale Anzahl von Punkten im virtuellen Raum platziert, beschrieben. Wir betrachten nur eine Hierarchie-Ebene, die der Zustandsmaschine `SetOfPointsInteractor`. Zur Veranschaulichung wird lediglich ein Teil der Definition besprochen und später der Aufbau der gesamten Definition in der Abbildung 4.9 überblicksweise dargestellt. Die Zustandsmaschine regelt die Interaktion auf eine begrenzte Liste von Punkten. Die Koordinaten der gesetzten Punkte werden in einem Objekt `PointListData` gehalten.

```
1  <stateMachine NAME="SetOfPointsInteractor">
2   <state NAME="nopoints" ID="1" START_STATE="TRUE">
3    <transition NAME="addPoint but check n" NEXT_STATE_ID="3" EVENT_ID="3"
4      SIDE_EFFECT_ID="SeCHECKNMINUS1" />
5   </state>
6
7   <state NAME="space left" ID="2">
8    <transition NAME="addPoint but check n" NEXT_STATE_ID="3" EVENT_ID="3"
9      SIDE_EFFECT_ID="SeCHECKNMINUS1" />
10  </state>
11
12  <state NAME="checkedN" ID="3">
13   <transition NAME="n smaler N" NEXT_STATE_ID="2" EVENT_ID="220"
14     SIDE_EFFECT_ID="SeADDELEMENT" />
15   <transition NAME="n greater equals N" NEXT_STATE_ID="4" EVENT_ID="221"
16     SIDE_EFFECT_ID="SeADDELEMENT" />
17  </state>
18
19  <state NAME="set full" ID="4">
20    ...
21  </state>
22
23 </stateMachine>
```

Abbildung 4.5: Definition einer N-Punkte-Zustandsmaschine

Die Abbildung 4.5 zeigt einen Auszug aus der XML-Datei. Der Aufbau der Datei wurde bereits in 3.2.4 beschrieben. Die Definition der Ereignisse, die in der Definition der Zustandsmaschine vorkommen, sind in Abbildung 4.6 aufgelistet. Die Abbildung 4.7 zeigt das entsprechende Statechart-Diagramm des Beispiels.

Die Maschine `SetOfPointsInteractor` bildet das Verhalten einer Liste von Punkten, die nur eine bestimmte Anzahl beinhalten darf, ab. Nehmen wir an, der Grenzwert liegt

bei drei Punkten (vgl. Beispiel einer Winkelvermessung, Abb. 2.9). Der Benutzer setzt der Reihe nach die Punkte und soll nach dem dritten Punkt keinen neuen mehr setzen dürfen. Dieses Verhalten wird durch vier Zustände (vgl. Abb. 4.7) realisiert.

```
1  <events STYLE="Powerpoint">
2   <event NAME="left MouseBN" ID="1" TYPE="MouseButtonPress"
3     BUTTON="LeftButton" BUTTONSTATE="NoButton" KEY="Key_none" />
4
5   <event NAME="left MouseBN + Shift"' ID="3" TYPE="MouseButtonPress"
6     BUTTON="LeftButton" BUTTONSTATE="ShiftButton" KEY="Key_none" />
7
8   <event NAME="Delete" ID="12" TYPE="KeyPress"
9     BUTTON="NoButton" BUTTONSTATE="NoButton" KEY="Key_Delete" />
10
11  <!--eigens ausgeloeste Events/intern-->
12  <event NAME="<N-1" ID="220" TYPE="User"
13    BUTTON="NoButton" BUTTONSTATE="NoButton" KEY="Key_none" />
14
15  <event NAME=">N-1" ID="221" TYPE="User"
16    BUTTON="NoButton" BUTTONSTATE="NoButton" KEY="Key_none" />
17 </events>
```

Abbildung 4.6: Definition der Ereignisse

Die Maschine startet im ersten Zustand `nopoints`. In diesem Zustand ist die Liste leer. Der Zustand besitzt einen Übergang in den dritten Zustand. `EventID = "3"` besagt, dass das Ereignis mit der $ID = 3$ (vgl. Abb. 4.6) den Übergang auslöst.

Das Ereignis enthält eine QT-eigene Konstante `MouseButtonPress`, die besagt, dass die Maus das Ereignis ausgelöst hat. `LeftButton` definiert näher, dass es die linke Maus-Taste war. Wenn ein so genannter *Modifier* während der Auslösung des Ereignisses gedrückt wurde, so wird dies in `BUTTONSTATE` hinterlegt, in unserem Beispiel ist es `ShiftButton`. `Key_none` besagt, dass keine weitere Taste gedrückt wurde.

Der Benutzer klickt nun mit der linken Maus-Taste auf einen Bereich in der Anwendung und drückt dabei die Shift-Taste. Die Maschine wechselt vom ersten in den dritten Zustand.

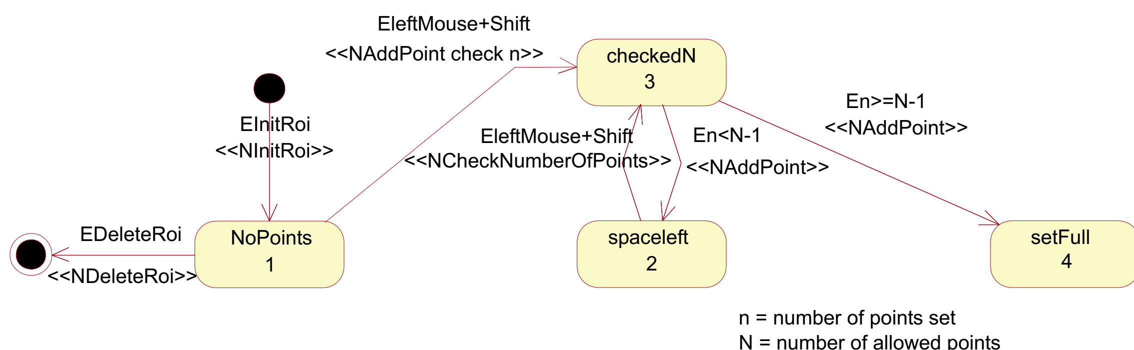


Abbildung 4.7: vereinfachter Aufbau der Zustandsmaschine SetOfPointsInteractor; StateChart-Diagramm

Der dritte Zustand ist ein Wächter-Zustand. In ihm wird überprüft, ob nach dem Hinzufügen des Punktes die Liste voll ist. Ist dies der Fall, so geht die Maschine in den vierten Zustand, ist dies nicht der Fall, geht sie in den zweiten über. Abbildung 4.8 zeigt einen Auszug aus der Klasse `SetOfPointsInteractor`. Nach dem Zustandswechsel wird die Aktion ausgeführt (vgl. Abb. 4.1), hier die Aktion `SeCHECKNMINUS1`. In der Methode `ExecuteSideEffect(..)` der Klasse `SetOfPointsInteractor` wird je nach Ergebnis der Überprüfung

$$n < N - 1$$

n = Anzahl Punkte in Liste
N = maximale Anzahl Punkte in Liste

ein neues Ereignis erzeugt, welches den nächsten Übergang spezifiziert. Die Liste ist noch leer, $0 < 3-1$ ist wahr, folglich wird ein Ereignis generiert, welches die Zustandsmaschine in den Zustand 2 versetzt. Nach dem Zustandswechsel wird der Punkt der Liste hinzugefügt.

Der zweite Zustand besagt, dass die Liste Punkte enthält, jedoch noch nicht voll ist. Erneutes Klicken der linken Maus-Taste kombiniert mit dem Drücken der Shift-Taste versetzt die Maschine in den dritten Zustand zurück. Die Bedingung $n < N - 1$ wird überprüft, mit dem Ergebnis, dass ein Ereignis generiert wird, das die Maschine abermals in den zweiten Zustand überführt. Beim nächsten Ereignis „linke-Maus-Taste+Shift“ ergibt die Überprüfung, dass nach dem Hinzufügen des Punktes in die Liste diese voll sein wird. Deshalb wird ein Ereignis generiert, welches die Maschine in den vierten Zustand versetzt. Nun kann der Benutzer keinen Punkt mehr hinzufügen, da in dem Zustand kein Ereignis „linke-Maus-Taste+Shift“ akzeptiert wird.

Natürlich wäre dieser Zustand fehlerhaft definiert, denn er würde sich gleich einem *toten Zustand* (vgl. 2.7.5) verhalten. Dieses Beispiel ist aber nur ein Auszug aus einem komplexeren Aufbau. Abbildung 4.9 zeigt die gesamte Maschine mit der Funktionalität des Löschens eines Punktes sowie mit der Möglichkeit, einen Punkt zu selektieren und zu verschieben.

4.2.6 Anpassung, Erweiterung und Neuentwicklung

Das MITK bietet bereits Standard-Klassen, mit denen ein Entwickler eine Anwendung implementieren kann. Spezifischere Komponenten müssen jedoch entwickelt werden. Im Laufe der Zeit werden mehr und mehr Klassen dem MITK hinzugefügt werden, so dass die Implementierung eigener Klassen seltener wird.

Auch für die Interaktion stehen schon Standard-Klassen bereit. Wird eine neue Interaktion benötigt, so kann es schon ausreichen, die Definition in der XML-Datei anzugleichen oder zu ergänzen. Kleinere Änderungen und auch Erweiterungen sind hierdurch möglich. Soll eine komplexere Interaktion mit neuen Aktionen modelliert werden, so muss eine neue Klasse von einer Zustandsmaschinen-Klasse abgeleitet werden. Hier bietet sich an, eine von

```
1  bool mitk::SetOfPointsInteractor::ExecuteSideEffect(int sideEffectId,
2    mitk::StateEvent const* stateEvent, int objectEventId, int groupEventId)
3  {
4   ...
5   switch (sideEffectId)
6   {
7    case SeCHECKNMINUS1:
8    {
9     if (pointList->GetSize()<(m_N-1))
10    {
11     mitk::StateEvent* newStateEvent =
12       new mitk::StateEvent(StSMALERNMINUS1, stateEvent->GetEvent());
13     this->HandleEvent( newStateEvent, objectEventId, groupEventId );
14     ok = true;
15    }
16    else
17    {
18     mitk::StateEvent* newStateEvent =
19       new mitk::StateEvent(StLARGERNMINUS1, stateEvent->GetEvent());
20     this->HandleEvent(newStateEvent, objectEventId, groupEventId );
21     ok = true;
22    }
23   }
24   break;
25
26   case SeADDELEMENT:
27   {
28    mitk::ITKPoint3D itkPoint;
29    mitk::vm2itk(posEvent->GetWorldPosition(), itkPoint);
30
31    this->UnselectAll(objectEventId, groupEventId);
32    int lastPosition = pointList->GetSize();
33    PointOperation* doOp =
34      new mitk::PointOperation(OpADD, itkPoint, lastPosition);
35    if (m_UndoEnabled)
36    {
37     PointOperation *undoOp =
38       new mitk::PointOperation(OpDELETE, itkPoint, lastPosition);
39     OperationEvent *operationEvent =
40       new OperationEvent(pointList, doOp, undoOp, objectEventId, groupEventId);
41     m_UndoController->SetOperationEvent(operationEvent);
42    }
43    pointList->ExecuteOperation(doOp);
44    ok = true;
45   }
46   break;
47  ...
```

Abbildung 4.8: Auszug der Methode ExecuteSideEffect der Klasse SetOfPointsInteractor

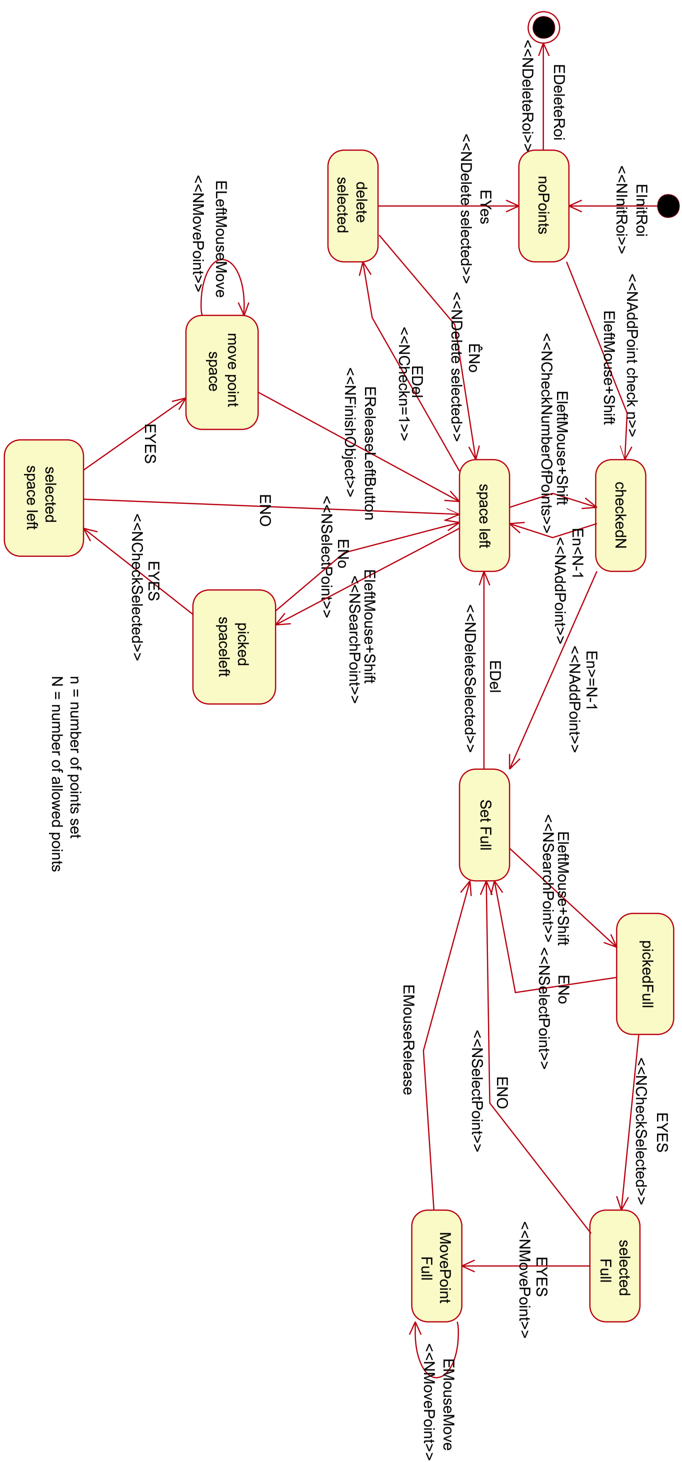


Abbildung 4.9: Statechart-Diagramm des Verhaltensmusters SetOfPointsInteractor

einer möglichst ähnlichen Klasse abzuleiten. Bereits implementierte Aktionen können so übernommen und durch einen entsprechenden Aufruf der Superklasse genutzt werden. Nur die hinzukommenden Aktionen werden implementiert. Die neu entstandene Klasse muss hierauf mit einem Verhaltensmuster, in dem die neuen Aktionen genutzt werden, kombiniert werden. Wird eine neue Daten-Klasse benötigt, so kann sie ebenfalls abgeleitet und ergänzt werden.

```
1  class AngleInteractor : public SetOfPointsInteractor
2  {...}
3
4  bool mitk::AngleInteractor::ExecuteSideEffect(int sideEffectId,
5    mitk::StateEvent const* stateEvent, int objectEventId, int groupEventId)
6  {
7   bool ok = SetOfPointsInteractor::ExecuteSideEffect(sideEffectId, stateEvent,
8     objektEventId, groupEventId);
9   if (ok)
10   return ok;
11  else
12   switch (sideEffectId)
13   {
14    case SeCHANGEORDER:
15    {
16     //generate operations, that change the order in the PointListData
17     ...
18    }
19    default:
20     ok = false;
21   }
22  return ok;
23 }
```

Abbildung 4.10: Beispiel für die Implementierung einer neuen Interaktion

Ein Beispiel: Die Interaktion des in Abbildung 2.9 gezeigten Werkzeugs zur Winkelvermessung soll dahingehend ergänzt werden, dass alle drei Winkel des Dreiecks, das durch die drei Punkte definiert wird, der Reihe nach angezeigt werden können. Durch eine bestimmte Tastenkombination, z.B. „Strg+W“, kann der angezeigte Winkel die Ecke wechseln. Von der zuvor genutzten Interaktions-Klasse `SetOfPointsInteractor` wird eine neue Klasse `AngleInteractor` abgeleitet. Lediglich die Methode `ExecuteSideEffect(..)` (vgl. Abb. 4.10) und ein Übergang in mehreren Zuständen des Verhaltensmusters müssen ergänzt werden.

4.3 Implementierung des Undo/Redo-Modells

Die Undo-Klassen sind bis auf die des eigentlichen Undo-Modells einfach aufgebaut. In der Klasse `UndoController` werden lediglich eine Liste mehrerer Undo-Modell-Objekte verwaltet und die Methodenaufrufe `Undo (..)` und `Redo(..)` an das ausgewählte Undo-Modell weitergereicht. Die eigentliche Logik des Undos liegt in der Klasse des Undo-Modells. Im Folgenden gehen wir deshalb kurz auf die Realisation des eingeschränkten linearen Undo-

Modells (vgl. 2.8.3) ein.

4.3.1 Eingeschränktes lineares Undo-Modell

Das Modell ist bereits implementiert und befindet sich erfolgreich im Einsatz.

```
1  bool mitk::LimitedLinearUndo::Undo(bool fine)
2  {
3   if(m_UndoList.size()<1)
4    return false;
5   if (fine == true)//undo all ObjectEventId
6   {
7    int undoObjectEventId = (m_UndoList.back())->GetObjectEventId();
8    do
9    {
10    OperationEvent* operationEvent = m_UndoList.back();
11    this->SwapOperations(operationEvent);
12    operationEvent->GetDestination()->ExecuteOperation(operationEvent->GetOperation()
       );
13
14    m_RedoList.push_back(operationEvent);
15    m_UndoList.pop_back();
16    if (m_UndoList.empty())
17     break;
18   } while ((m_UndoList.back())->GetObjectEventId() == undoObjectEventId);
19  }
20  else //fine==false so undo all GroupEventId
21  {
22   int undoGroupEventId = (m_UndoList.back())->GetGroupEventId();
23   do
24   {
25    OperationEvent* operationEvent = m_UndoList.back();
26    this->SwapOperations(operationEvent);
27    operationEvent->GetDestination()->ExecuteOperation(operationEvent->GetOperation()
       );
28    m_RedoList.push_back(operationEvent);
29    m_UndoList.pop_back();
30    if (m_UndoList.empty())
31    {
32     break;
33    }
34   } while ((m_UndoList.back())->GetGroupEventId() == undoGroupEventId);
35  }
36  return true;
37 }
```

Abbildung 4.11: Die Methode Undo(..) der Klasse LimitedLinearUndo

Der Benutzer hat die Möglichkeit bei einem Undo zu entscheiden, ob er einen Befehl oder eine Gruppe von zusammengehörigen Befehlen rückgängig machen will (vgl. 3.3.6). Der Methode `Undo(..)` (vgl. Abb. 4.11) wird deshalb ein boolscher Parameter mit übergeben, der spezifiziert, für welche Möglichkeit sich der Benutzer entschieden hat.

In der Methode wird zunächst unterschieden, für welche Variante sich der Benutzer entschieden hat. Ist der Wert des übergebenen Parameters `fine` gleich `true`, so soll ein einziger Befehl rückgängig gemacht werden. Die Nummer `ObjectEventId` des obersten Objekts der Befehlsgeschichte wird temporär gespeichert, um zu realisieren, dass alle Objekte, die die

gleiche Nummer wie das zuerst entnommene Element haben, aus der Liste herausgeholt werden. Danach wird eine Schleife so lang ausgeführt, bis sich die Nummer `ObjektEventId` des obersten Objekts der Liste von der gespeicherten Nummer unterscheidet. In der Schleife wird die Methode `SwapOperations(..)` jedes gefundenen Objekts ausgeführt. In dieser Methode werden die beiden Referenzen der Operations-Objekte vertauscht (vgl. 3.3.3). Im Anschluss wird das Operations-Objekt, nach `SwapOperations(..)` ist dies das Objekt `undoOperation`, an das in `m_Destination` referenzierte Daten-Objekt geschickt. Das Daten-Objekt führt die übergebene Operation aus. Nun wird das Objekt von der Undo-Befehlsgeschichte in die Redo-Befehlsgeschichte transferiert. Ist die Undo-Liste hierauf leer, so wird die Ausführung der Schleife abgebrochen.

Ähnlich ist auch die Schleife für das Rückgängigmachen einer Gruppe von Operationen aufgebaut. Hier wird die Schleife jedoch erst dann verlassen, wenn die Nummer `GroupEventId` des obersten Objekts der Befehlsgeschichte wechselt.

Das eingeschränkte lineare Undo-Modell hat sich auch im Rahmen des MITKs als benutzerfreundlich herausgestellt. Ein Undo kann auch ohne Information über die gewünschte Granularität ausgeführt werden. Die Methode `Undo(..)` ist *überladen* deklariert, d.h. es existieren zwei Methoden namens Undo. Die Methode mit dem Parameter `fine` wurde oben beschrieben. In einer weiteren Methode wird die oben beschriebene mit dem Parameter `fine` gleich `true` aufgerufen, so dass sich das Undo-Modell so verhält, als würde es die Funktionalität des „groben“ Undos nicht bieten. Der Entwickler entscheidet über das Angebot der Funktionalität.

Kapitel 5

Ergebnis

In dieser Arbeit wurde ein generisches Interaktionsmodell mit Undo-Funktionalität entworfen und realisiert. Das Modell ist ein wichtiger Bestandteil des Medical Imaging Toolkits und ermöglicht die Interaktion mit Rücknahmemöglichkeit. Es erleichtert die Entwicklung von interaktiven Bildverarbeitungsprogrammen in der Medizin.

In der Arbeit wurde das Augenmerk besonders auf Benutzerfreundlichkeit, Flexibilität, Wartbarkeit, Übersichtlichkeit und Ressourcen-Sparsamkeit gelegt. Die Benutzerfreundlichkeit der mit dem MITK entwickelten Anwendungen wurde über die Beachtung der acht goldenen Regeln zur Erstellung von Benutzerschnittstellen (vgl. 2.5) gefördert. Das Angebot von Standard-Interaktionskomponenten (z.B. Interaktion auf einen Punkt) fördert die Konsistenz der Interaktion einer Anwendung. Ferner sehen die Standard-Interaktionskomponenten den Gebrauch der üblichen Shortcuts vor. Über diese Komponenten sind ebenso die Dialoge mit definierten Abschlüssen sowie die Realisation vorhersehbarer Systemreaktionen möglich. Durch die mit Sorgfalt entwickelten Standardkomponenten werden Fehler vermieden. Wird trotzdem ein Fehler produziert, so wird dieser, wenn möglich, vom System zurückgenommen (vgl. Abb. 4.1 Zeile 27).

Bis auf wenige Ausnahmen hat der Benutzer die Möglichkeit, einen von ihm gegebenen Befehl rückgängig zu machen und danach zu wiederholen. In Ausnahmefällen, in denen ein Befehl nicht undo-fähig ist, wird der Benutzer vor der Ausführung darüber in Kenntnis gesetzt. Das Undo-Konzept sieht die Gruppierung von Befehlen vor, die zusammen rückgängig gemacht und wiederholt werden können. Somit kann noch genauer auf die Wünsche des Benutzers eingegangen werden.

Dem Entwickler sind alle möglichen Freiheiten gelassen, die Interaktion zu modellieren. Durch die interne Definition von Ereignissen können neue Eingabegeräte für die Interaktion genutzt werden. Der generische Ansatz des Interaktionsmodells erlaubt schnelle Änderungen an der Interaktion sowie ein Angebot von mehreren Interaktions-Schemata. Die Übersichtlichkeit wird über eine zentrale Definition der Interaktion gesteigert. Gefördert wird hierdurch auch die Wartbarkeit einer Anwendung.

Die Interaktion wird in mehrere Hierarchien unterteilt, was der Interaktion die Komplexität nimmt. Weiterhin kann sie in einer XML-Datei auf einem gewohnten Abstraktionslevel

modelliert werden.

Ebenso wichtig ist die Minimierung der Ressourcen durch Wiederverwendung der Verhaltensmuster.

Alle nötigen Interaktionsklassen sowie einige Standard-Interaktionskomponenten sind implementiert. Die in 1.2 beschriebenen Anforderungen wurden erfüllt.

Die Notwendigkeit, neue Interaktionskomponenten zu entwickeln, wird sich verringern. Zwar wird zunächst noch gewohnt viel implementiert werden, doch kann im Laufe der Zeit durch Wiederverwendung die Entwicklungszeit immer mehr minimiert werden.

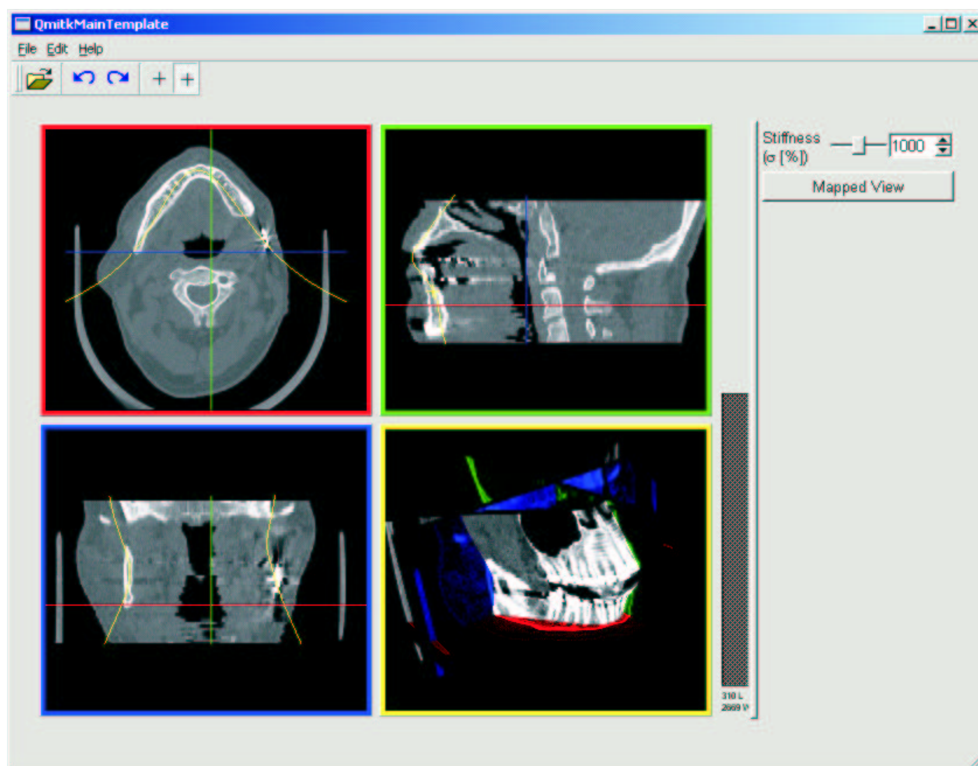


Abbildung 5.1: Applikation für die Erstellung einer Ansicht auf eine gekrümmte Oberfläche; dreidimensionale Ansicht (gelb) zeigt einen Schnitt durch die Zähne

Das Resultat der Arbeit wird bereits in mehreren Anwendungen der medizinischen Forschung erfolgreich eingesetzt. Darunter zählen Applikationen für die interaktive Benutzung der ITK-Registrierungs-Algorithmen, für die Volumetrie von Herz-Kavitäten, für das Setzen von Landmarken in bildgesteuerten Prozeduren und für die Erstellung einer Ansicht auf eine gekrümmte Oberfläche, die durch eine Menge von Punkten definiert wird. Abbildung 5.1 zeigt die zuletzt genannte Applikation. Durch das interaktive Setzen von Punkten in den zweidimensionalen Ansichten (rot, grün, blau) wird eine Oberfläche bestimmt. Diese wird in den zweidimensionalen Ansichten als gelbe Linie(n) und in der dreidimensionalen Ansicht (gelb) als Schnitt durch die Zähne dargestellt. In der Abbildung 5.2 ist die gekrümmte

Oberfläche zweidimensional (aufgeklappt) dargestellt. Die einzelnen Punkte (gelb) zeigen die interaktiv gesetzten Stützstellen der Oberfläche.

Durch das generische Interaktionskonzept können die Punkte interaktiv gesetzt, verschoben und gelöscht werden. Durch das Undo-Konzept können die Befehle vereinzelt oder in Gruppen rückgängig gemacht oder wiederholt werden. Mehrfach reduzierten die hier vorgestellten Konzepte die Entwicklungszeit der Anwendungen. Somit konnte deutlicher auf die tatsächliche Aufgabenstellung der Entwicklung eingegangen werden, ohne unnötige Arbeit bei der Erstellung einer Entwicklungsumgebung zu verschwenden.

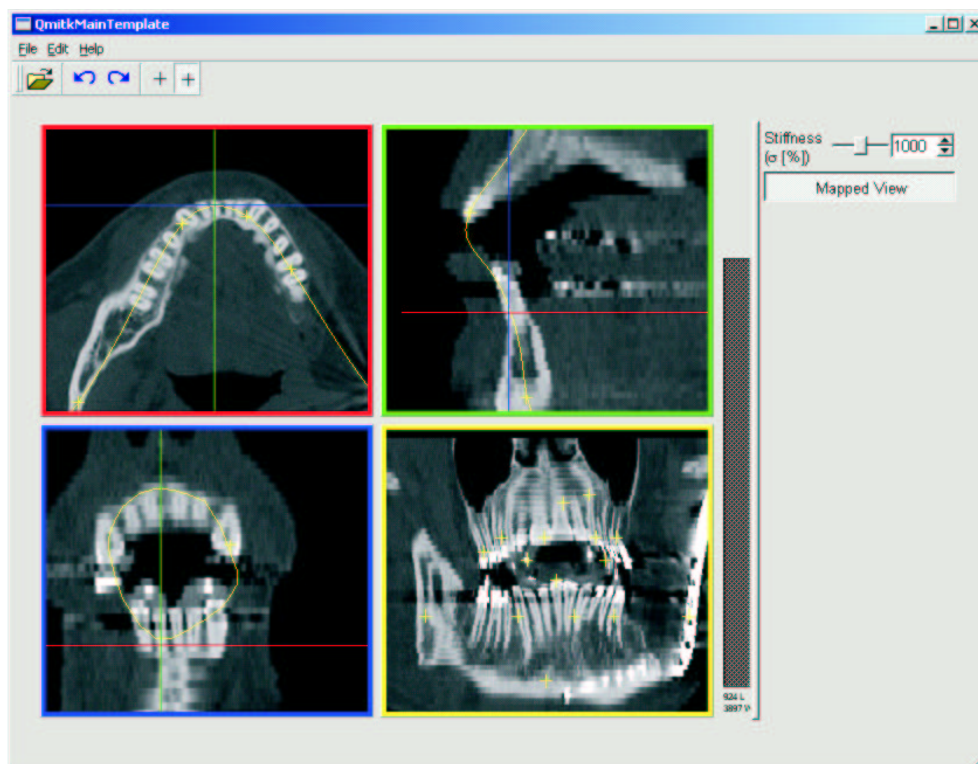


Abbildung 5.2: gleiche Applikation und gleicher Datensatz wie in Abb. 5.1; zweidimensionale Ansicht (gelb, rechts unten) zeigt die aufgespannte Oberfläche

Für die Konferenz *Medical Imaging 2004* der Sozietät *SPIE - The International Society for Optical Engineering* in San Diego, CA USA, wurde ein Beitrag mit dem Titel „The Medical Imaging Interaction Toolkit (MITK) - a toolkit facilitating the creation of interactive software by extending VTK and ITK“ eingereicht. Er beschreibt Zielsetzungen, Methoden und Ergebnisse des MITKs und zeigt Bilder möglicher Anwendungen.

Die Veröffentlichung des Software-Pakets ist für das Frühjahr 2004 geplant. Ab dem Zeitpunkt wird auch die Dokumentation aller Klassen online einsehbar sein. Die offizielle MITK-Web-Seite liegt unter folgender Adresse: www.mitk.org

Kapitel 6

Diskussion und Ausblick

Gegenstand dieses Kapitels ist es, die Sinnhaftigkeit des generischen Interaktionskonzepts mit Undo-Funktionalität und dessen Anwendbarkeit in der Praxis kritisch zu beleuchten. Zudem wird ein Ausblick auf weitere Arbeiten geboten.

Das in dieser Arbeit entstandene Konzept reiht sich zusammen mit den anderen MITK-Ansätzen als Toolkit in die Vielzahl der medizinischen Software-Pakete ein. Jedoch steht es so gut wie alleine da, wenn man nur die medizinischen Bildverarbeitungsprogramme mit Undo betrachtet. Dabei ist der Nutzen einer Undo-Funktionalität besonders bei der Bildverarbeitung unumstritten. Eine weitere Besonderheit des MITKs ist die Vereinigung von ITK und VTK, von denen jedes ständig weiterentwickelt wird. Auf dem Gebiet der Medizin ist die Mensch-Computer-Interaktion sehr komplex. Neuartige Eingabegeräte erhöhen die Schwierigkeiten. Durch die eigene Definition der Ereignisse im MITK können diese Probleme geschwächt und durch die Modellierung der Interaktion in Teilinteraktionen beseitigt werden.

Kritisch betrachtet werden muss die eventuell längere Einarbeitungszeit in das MITK. Die Klassen sind objektorientiert in der weit verbreiteten Programmiersprache C++ geschrieben. Der Umgang mit den Klassen des generischen Interaktionskonzepts kann ungewohnt erscheinen. Zustandsmaschinen werden jedoch in allen Informatik-Studiengängen besprochen und so stellt die Modellierung der Interaktion ein geringes Problem dar.

Auch das Undo-Konzept sollte keine größeren Schwierigkeiten bereiten, da sich der Entwickler kaum in die Methodik des Undos einarbeiten muss. Einzig die Erstellung der inversen Operationen und das Senden der Informationen an das Objekt `undoController` müssen bewerkstelligt werden. Einmal eingearbeitet stehen dem Entwickler alle Vorzüge von ITK, VTK und MITK zur Verfügung.

Die Definition der Interaktion in der XML-Datei soll ebenso kritisch betrachtet werden. Von der sofortigen Modellierung in der XML-Datei ist abzuraten, da die schriftliche Form in Sachen Übersichtlichkeit der grafischen Form unterlegen ist. Vielmehr sollte die Planung der Interaktion in einem Modellierungsprogramm wie z.B. *IBMs Rational Rose* vorgenommen werden. In Aussicht steht ein eigenes MITK-Modellierungsprogramm der Interaktion. Hiermit soll die grafische Definition von Interaktion anstelle der schriftlichen möglich sein.

Das Programm erstellt nach der grafischen Modellierung selbstständig die schriftliche Definition in der XML-Datei. Bereits bestehende Interaktionskomponenten, die nach einer gewünschten Anpassung in das Konzept übernommen werden können, sollen zur Verwendung angeboten werden.

Eine denkbare Erweiterung wäre die Verlagerung der Benutzerausgaben aus den Interaktionsklassen in ein eigenes Fehler-Management-System. In den Interaktionsklassen könnte im Falle eines Fehlers eine Nachricht an das Fehler-Management-System mit der Bezeichnung des Fehlers oder einer Identifikationsnummer geschickt werden. In diesem System könnte dann die Ausgabe der Fehlermeldung angepasst werden. Ein unterschiedlicher Informationsgehalt sowie ein unterschiedliches Layout könnten vom Entwickler bestimmt werden. Standardmäßig könnten die Ausgaben in einer bestimmten Form angeboten und je nach Wunsch des Entwicklers abgeändert werden. Somit soll Mehrarbeit verhindert, jedoch Flexibilität der Benutzerausgaben gefördert werden.

In Bezug auf das Undo-Konzept ist zu empfehlen, weitere Undo-Modelle nach Bedarf zu implementieren. Das eingeschränkte lineare Undo-Modell genügt bisher den Anforderungen an die Rücknahmemöglichkeit, jedoch könnte ein spezielleres Undo-Modell, wie etwa das baumbasierte lineare Undo-Modell, einen weiteren Nutzen bieten. Beispielsweise wäre es vorteilhaft, wenn in einer komplexen Anwendung auf keinen Fall die Befehlsgeschichte gelöscht werden darf.

Zusammenfassend lässt sich sagen, dass die entwickelten Konzepte die Erstellung medizinischer Bildverarbeitungssysteme erleichtern und beschleunigen. Durch den Einsatz von Standard-Interaktionen und Undo kann die Benutzerfreundlichkeit der Anwendung gesteigert werden.

Abbildungsverzeichnis

2.1 *links*: Zahlenschloss; *rechts*: Kombinationsschloss an einem victorianischen Safe 11

2.2 Die vier Elemente einer Zustandsmaschine (traditionell): Zustand mit Nummer (rot), Übergang (schwarz), Ereignis (grün) und Aktion (blau) 11

2.3 Die unterschiedlichen Diagramm-Notationen 13

2.4 *links*: Zustandsübergangsdiagramm (traditionell); *rechts*: zugehörige state-to-state-Tabelle 14

2.5 Fehler in Zustandsmaschinen 15

2.6 Fehler in einer deterministischen Zustandsmaschine 15

2.7 Zustandsmaschinen-Modell eines Kellerspeichers; *links*: nichtdeterministischer Ansatz; *rechts*: deterministischer Ansatz mit Wächtern 16

2.8 Realisierung von Wächtern mittels Zwischenzuständen 17

2.9 Beispiel einer Winkelvermessung 18

2.10 Einsatz eines Geraden- und eines Bezier-Werkzeugs; Tangente der Bezier-Kurve ist rot-gestrichelt dargestellt 19

2.11 Lineares Löschverfahren bei Speicherüberlauf; jeder Kreis symbolisiert ein Operationsobjekt 24

2.12 Baumbasiertes Löschverfahren bei Speicherüberlauf; eine Regel oder der Benutzer entscheidet über den zu löschenden Ast 25

2.13 Uneingeschränktes lineares Undo-Modell; die Reihenfolge der Operationen ist farblich gekennzeichnet und nummeriert 26

2.14 Lineares Undo-Modell mit Befehlsbaum 27

3.1 Klassen-Diagramm StateMachine 35

3.2 Sequenz-Diagramm Zustandsmaschine 36

3.3 Klassen-Diagramm State 37

3.4 Klassen-Diagramm Transition 37

3.5 Objekt-Diagramm der Zustandsmaschine; Aufbau und Referenzierung des Start-Zustands durch ein Objekt der Klasse StateMachineFactory 38

3.6 Klassen-Diagramm StateMachineFactory 39

3.7 Hierarchischer Aufbau der Zustandsmaschinen in der XML-Datei 40

3.8 Teilung einer Zustandsmaschine in Logik und Verhaltensmuster 41
3.9 Zugriff mehrerer Zustandslogiken auf ein Zustandsmuster 41
3.10 Beispiel einer Hierarchiebildung von Interaktionen 42
3.11 Modellierung der Interaktion in Abstraktionsebenen 43
3.12 Übersicht der von StateMachine abgeleiteten Klassen 44
3.13 Definition der Ereignisse in der XML-Datei 46
3.14 Klassen-Diagramm Event 48
3.15 Klassen-Diagramm PositionEvent 48
3.16 Klassen-Diagramm EventDescription 49
3.17 Klassen-Diagramm EventMapper 50
3.18 Klassen-Diagramm StateEvent 50
3.19 Klassen-Diagramm eines Datenbaum-Knotens 51
3.20 Übersicht der wichtigsten Interaktions-Klassen 52
3.21 Klassen-Diagramm der Operations-Klassen 56
3.22 Klassen-Diagramm OperationEvent 57
3.23 Objekt-Diagramm der Kommunikation zwischen Interaktions- und Daten-Objekten; *roter Pfeil*: Referenz; *blau gestrichelt*: Nachricht 58
3.24 Klassen-Diagramm UndoController 58
3.25 Übersicht der wichtigsten Undo-Klassen 62

4.1 Methode HandleEvent der Klasse StateMachine 64
4.2 Methode ConnectStates der Klasse StateMachineFactory und Methode ConnectTransitions der Klasse State 67
4.3 Testen einer Zustandsmaschine; die Methode parse der Interaktions-Klasse StateMachineFactory 68
4.4 Wiederverwendung der Interaktionsmuster 69
4.5 Definition einer N-Punkte-Zustandsmaschine 70
4.6 Definition der Ereignisse 71
4.7 vereinfachter Aufbau der Zustandsmaschine SetOfPointsInteractor; StateChart-Diagramm 71
4.8 Auszug der Methode ExecuteSideEffect der Klasse SetOfPointsInteractor . 73
4.9 Statechart-Diagramm des Verhaltensmusters SetOfPointsInteractor 74
4.10 Beispiel für die Implementierung einer neuen Interaktion 75
4.11 Die Methode Undo(..) der Klasse LimitedLinearUndo 76

5.1 Applikation für die Erstellung einer Ansicht auf eine gekrümmte Oberfläche; dreidimensionale Ansicht (gelb) zeigt einen Schnitt durch die Zähne 79
5.2 gleiche Applikation und gleicher Datensatz wie in Abb. 5.1; zweidimensionale Ansicht (gelb, rechts unten) zeigt die aufgespannte Oberfläche 80

Tabellenverzeichnis

2.1 Beispiel zum Ablauf eines eingeschränkten linearen Undo-Modells [Ost03]; a-e: Befehle, U: Undo, R: Redo, |: Position des Befehlszeigers 25
2.2 Beispiel zum Ablauf eines uneingeschränkten linearen Undo-Modells 26
2.3 Beispiel zum Ablauf eines linearen Undo-Modells mit Befehlsbaum; * symbolisiert eine Entscheidungssituation 27
2.4 Beispiele für inverse Operationen 30

Glossar

Applikation (*EDV*) Anwendung, Programm; (*Med.*) Verabreichung (von Medikamenten), Anwendung (von Heilverfahren)

Befehl Anweisung an das Programm zur Ausführung bestimmter Operationen

Befehlsgeschichte Element eines Undo-Modells, welches Informationen über ausgeführte Operationen festhält

Befehlszeiger Zeiger auf eine Stelle in der Befehlsgeschichte

boolscherWert der Wert kann entweder wahr oder falsch sein, 1 oder 0

Designphase Transformation des abstrakten Software-Modells aus dem Problemraum (Analysephase) in ein abstraktes Modell im Lösungsraum; die Entwurfsphase eines Softwareprodukts, die nach Ablauf in die Implementierungsphase mündet

Get − Methode Methode eines Objekts, welche (geschützt) eine Membervariable zurückliefert

GUI graphical user interface; grafische Benutzeroberfläche

interaktiv Dialog zwischen Anwender und Computer

Interface Schnittstelle

Intervention medizinischer Eingriff

intraoperativ (*Med.*) während der Operation

invasiv (*Med.*) eindringend

inverseOperation eine Operation, die ihre zugehörige Operation aufheben oder umkehren kann

inverserBefehl ein Befehl, der eine vom Benutzer ausgeführte Kette von Operationen aufheben oder umkehren kann

Kavität (*Med.*) Hohlraum

Kellerspeicher Speicher, in dem nur an einem Ende ein Element hinzugefügt und gelöscht werden kann

Membervariable eine veränderliche Größe eines bestimmten Datentyps, die von einem Objekt umschlossen wird

Meta − Befehl der über einer Operation befindliche Befehl; Undo- oder Redo-Befehl

Modifier (*engl.*) Modifikator; Taste, die kein Ereignis auslösen, jedoch ein ausgelöstes Ereignis näher spezifizieren kann; z.B. Shift, Alt, Strg, Meta

Multithreading das Nutzen mehrerer paralleler Prozesse

Navigation die Einhaltung des gewählten Kurses und die Standortbestimmung

Null − Aktion Φ, Kennzeichnung eines Übergangs, der keine Aktion zur Folge hat

Open − Source Software, deren Quellcode veröffentlicht wurde und an dem freie Programmierer arbeiten können; LINUX und IntelliCAD sind typische Open-Source-Projekte

Operation (*EDV*) ein Rechenvorgang; (*Med.*) chirurgischer Eingriff in den Organismus

Phantom 3D-Eingabegerät; einem Skalpell nachempfunden

Pixel Punkt in einem Bild (zweidimensional)

präoperativ vor der Operation

Primitive geometrische Grundform; z.B. Punkt, Linie, Polygon

Quellcode (*engl.*) source code; Programmcode; implementierter Algorithmus

RationalRose objektorientierte, grafische UML-Software-Entwicklungsplattform; übernommen von IBM; http://www.rational.com/

Redo eine bereits rückgängig gemachte Operation erneut ausführen

Registrierung Prozess der Ausrichtung zweier Datensätze; Korrespondenzen finden; z.B. einen CT-Datensatz mit einem MR-Datensatz vereinen, um die Daten beider auswerten zu können

Schichtaufnahmen die von einem med. bildgebenen Verfahren (CT, PET etc.) gewonnenen 2D-Schichten; die Menge aller Schichtbilder eines Datensatzes ergibt ein Volumen

Scrollen Ansichtsbereich verändern; (*umgangssprachlich*) Bewegung mit dem Mausrad

Segmentierung Prozess der Identifizierung und Klassifizierung digital aufgenommener Strukturen

Shortcut Tastenkombination, die einen bestimmten Befehl auslöst

UML Unified Modeling Language; http://www.uml.org/

Undo eine ausgeführte Operation rückgängig machen; Auswirkungen einer Operation aufheben

Undo – Konzept Entwurf einer Undo-Realisation

Undo – Modell erlaubt dem Benutzer, Befehle/Operationen nach definierten Regeln rückgängig zu machen

Voxel Punkt in einem Volumen (dreidimensional)

Literaturverzeichnis

[Aro91] Barry Arons. Hyperspeech: Navigating in speech-only hypermedia. In *(UK) Conference on Hypertext*, pages 133–146, 1991.

[Ash56] W.R. Ashby. *An introduction to cybernetics.* Chapman and Hall, 1956.

[Ber94] T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

[BG87] P.S. Brown and J.D. Gould. An experimental study of people creating spreadsheets. *ACM Transactions on Office information Systems*, 5(3):258–272, 1987.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[Boe86] Barry W. Boehm. *A Spiral Model of Software Development and Enhancement*, volume 11. Software Engineering Notes, 1986.

[Car92] Tom Cargil. *C++ Programming Style.* Addison Wesley, 1992.

[CE00] Krzysztof Czarnecki and U.W. Eisenecker. *Generative Programming: methods, tool, and applications.* Addison-Wesley, 2000.

[Ced02] Per Cederqvist. *Version Management with CVS.* Network Theory Ltd., http://www.cvshome.org/docs/manual/, 2002. ISBN:0954161718.

[CMN80] S. Card, T. Moran, and A. Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, 1980.

[Con03] World Wide Web Consortium. *eXtensible Markup Language.* http://w3c.org/XML/, 2003.

[Dah99] M.K. Dahlheimer. *Programming with QT.* O'Reilly Verlag, 1999.

[Dou98] B.P. Douglass. *Real-time UML: developing efficient objects for embedded systems.* Addison Wesley, 1998.

[Far02] Gerald Farin. *Curves and Surfaces for CAGD; A Practical Guide.* Academic Press; Morgan Kaufmann Publishers, fifth edition, 2002.

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissdes. *Entwurfsmuster, Elemente wiederverwendbarer objektorientierter Software.* Bonn, Addison-Wesley, 1996.

[Gro03] Object Management Group. *Unified Modeling Laguage*^{TM}. http://www.uml.org/, 2003.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(4):514–529, 1988.

[HC91] J. Hooper and R. Chester. *Software reuse: guidelines and methods.* Plenum Press, 1991.

[Hei03] Tobias Heimann. Optimierung des segmentierungsvorgangs und evaluation der ergebnisse in der medizinischen bildverarbeitung. Diplomarbeit, Universität Heidelberg / Fachhochschule Heilbronn, 2003.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HU79] J.E. Hopcroft and S.D. Ullmann. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[JCJv92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering.* Addison Wesley, 1992.

[JCS84] J.E. Archer Jr, R. Conway, and F.B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, 1984.

[JJ91] R.E. Johnson and J.Zweig. Delegation in c++. *Journal of Object-Oriented Programming*, 4(11), 1991.

[Joh92] Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 1992.

[Jr86] G.B.Leemann Jr. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87, 1986.

[KDDS96] B. Klöppel, T. Dapper, C. Dietrich, and R. Seeber. *Objektorientierte Modellierung und Programmierung mit C++; Band 1: Grundkonzepte und praktischer Einsatz*. Oldenburg Verlag, 1996.

[Kun00] Tobias Kunert. Interaktive segmentierung von zwei- und dreidimensionalen datensätzen mit hilfe von aktiven konturen. *Technischer Bericht 118*, 2000.

[Mac95] L. Macauly. *Human-Computer Interaction for Software Designers*. International Thompson Comp. Press, 1995.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34:1045–1079, 1955.

[Min67] Marvin L. Minsky. *Computation - Finite and Infinte Machines*. Prentice-Hall Inc. Englewood Cliffs, 1967.

[MK96] B.A. Myes and D. S. Kosbie. Reusable hierachical command objects. In *Conference on Human Factors in Computing Systems*, pages 260–267, Vancouver, Canada, 14-18.April 1996. CHI'96, ACM-Press.

[Moo56] Edward P. Moore. Experiments on sequential machines. *In Automata studies: annals of mathematical studies, Princeton, N.J.: Princeton University Press*, 1956.

[MS94] T.M. Massie and J.K. Salisbury. The phantom haptic interface: A device for probing virtual objects. In *Dynamic Systems and Control 1994*, volume 1, pages 295–301, Chicago, 6-11.Nov 1994. ASME Haptic Interfaces for Virtual Environment and Teleoperator Systems 1994.

[Ola99] S.D. Olabarriaga. *Human-Computer Interaction for the Segmentation of Medical Images*. Doktorarbeit, Universität Amsterdam, 1999.

[OPB+97] R. O'Tool, R. Playter, W. Blank, N. Cornelius, W. Roberts, and M. Raibert. A novel virtual reality surgical trainer with force feedback: Surgeon vs. medical student performance. In *The Second PHANToM User's Group Proceedings*, 1997.

[Ost03] B. Ostermann. Softwareaktionen wiederholen und rückgängigmachen, 2003.

[PJ95] Meilir Page-Jones. *What ever programmer should know about object-oriented design*. New York: Dorset House, 1995.

[PK94] A. Prakash and M.J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, 1994.

[Pre94] J. Preece. *Human-Computer Interaction.* Addison-Wesleys, 1994.

[Pro03a] The GNU Project. Gnu emacs manual, 2003.

[Pro03b] Protos Software GmbH, http://www.protos.de/Spheres/Trice_D.html. *Trice*, 2003.

[Qua98] T. Quatrani. *Visual Modeling with Rational Rose and UML.* Addison Wesley, 1998.

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

[Rei85] W. Reisig. *Petri nets - an introduction.* Springer Verlag, 1985. EATCS Monographs on Theoretical Computer Science 4.

[Rob76] F.S. Roberts. *Descrete mathematical models. With applications to social, biological, and enviromental problems.* NJ: Prentice-Hall, 1976.

[S+95] A. Shebanow et al. *The Power of Frameworks.* Addison Wesley, 1995.

[Sal85] A. Salomaa. *Computation and automata.* Cambridge University Press, 1985.

[Sch00] A. Schmidt. Implicit human computer interaction through context. In *Personal Technologies*, volume 4, 2000.

[SKZ95] Michael Stark, Markus Kohler, and Projektgruppe (ZYKLOP). Video based gesture recognition for human computer interaction. Technical Report 593/1995, Universität Dortmund, 44221 Dortmund, GERMANY, citeseer.nj.nec.com/stark95video.html, 1995.

[Tei78] W. Teitelman. *Interlisp Reference Manual.* Xerox PARC, 1978.

[Thi90] H. Thimbleyby. *User Interface Design.* New York, ACM Press, 1990.

[Tro03a] Trolltech. *QT overview.* http://www.trolltech.com/products/qt/index.html, 2003.

[Tro03b] Trolltech. *QT's classes Documentation.* http://doc.trolltech.com/3.2/classes.html, 2003.

[Vit84] J.S. Vitter. Us&r: A new framework for redoing. *ACM SIGPLAN Notices*, 19(5):168–176, 1984.

[VL90] J.M. Vlissides and M.A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems, S237-268*, 8(3), 1990.

[VWH+03] M. Vetter, I. Wolf, P. Hassenpflug, M. Hastenteufel, R. Ludwig, L. Grenacher, G. Richter, W. Uhl, M. Büchler, and H. Meinzer. Navigation aids and real-time deformation modeling for open liver surgery. In *Visualization, Image-Guided Procedures, and Display*, volume 5029. SPIE, March 2003.

[Yan88] Yiya Yang. Undo support models. *International Journal of Man-Machine Studies*, 28(5):457–481, 1988.

Index

Übergang, 12, 37
  Null-, 45

Abbruch-Funktion, 22
Abstraktionsebene, 43
acht goldene Regeln, 8, 78
Aktion, 12, 43

Befehl
  -sgeschichte, 23
  -szeiger, 23
  inverser, 30
  Meta-, 28
Benutzerfreundlichkeit, 8

Datenbaum, 51
Designphase, 30

Ereignis, 12

Flucht-Funktion, 22

generische Programmierung, 7

Hierarchie, 41

Interaktion, 22, 72, 78
ITK, 1, 2, 6

Kellerspeicher, 16
Kommunikation über Objekte, 55

MITK, 5

Multithreading, 6

Navigation, 3

Open-Source, 6, 7
Operation
  inverse, 29, 54

QT, 2, 46, 66
Quellcode, 7, 20

Registrierung, 3, 6

Segmentierung, 3, 6, 10
Spiralmodell, 32
stabile Ausführungseigenschaft, 23
Strategie des kompletten erneuten Ausführens, 29, 54

tote Schleife, 15

UML, 16
Undo-Modell
  eingeschränktes lineares, 25, 28, 59, 76, 82
  History-, 28
  lineares mit Befehlsbaum, 27
  Single-, 24
  Skript-, 28
  Truncate/Reappend-, 28
  uneingeschränktes lineares, 23, 26
  US&R-, 28

Vererbung, 44
VTK, 1, 2, 7

Wächter, 16, 44, 72
Wiederverwendung, 19, 40, 69

Zustand, 11, 36
    finaler, 12
    initialer, 12, 16
    magischer, 15
    toter, 15
Zustandsmaschine, 10
    deterministisch, 11
    finite, 11
    Logik, 40
    nach Mealy, 13, 34
    nach Moore, 13, 34
    objektbasierte, 20, 34
    tabellenbasierte, 20, 34
    Verhaltensmuster, 40