
Programmtechnischer Ansatz für Effizienz und Verständlichkeit von Bausoftware in weltweiten Netzen

Dissertation zur Erlangung des akademischen Grades

Doktor-Ingenieur

im Fachbereich D, Abteilung Bauingenieurwesen
der Bergischen Universität Wuppertal



am Lehr- und Forschungsgebiet für
Theoretische Methoden und Angewandte Informatik

von

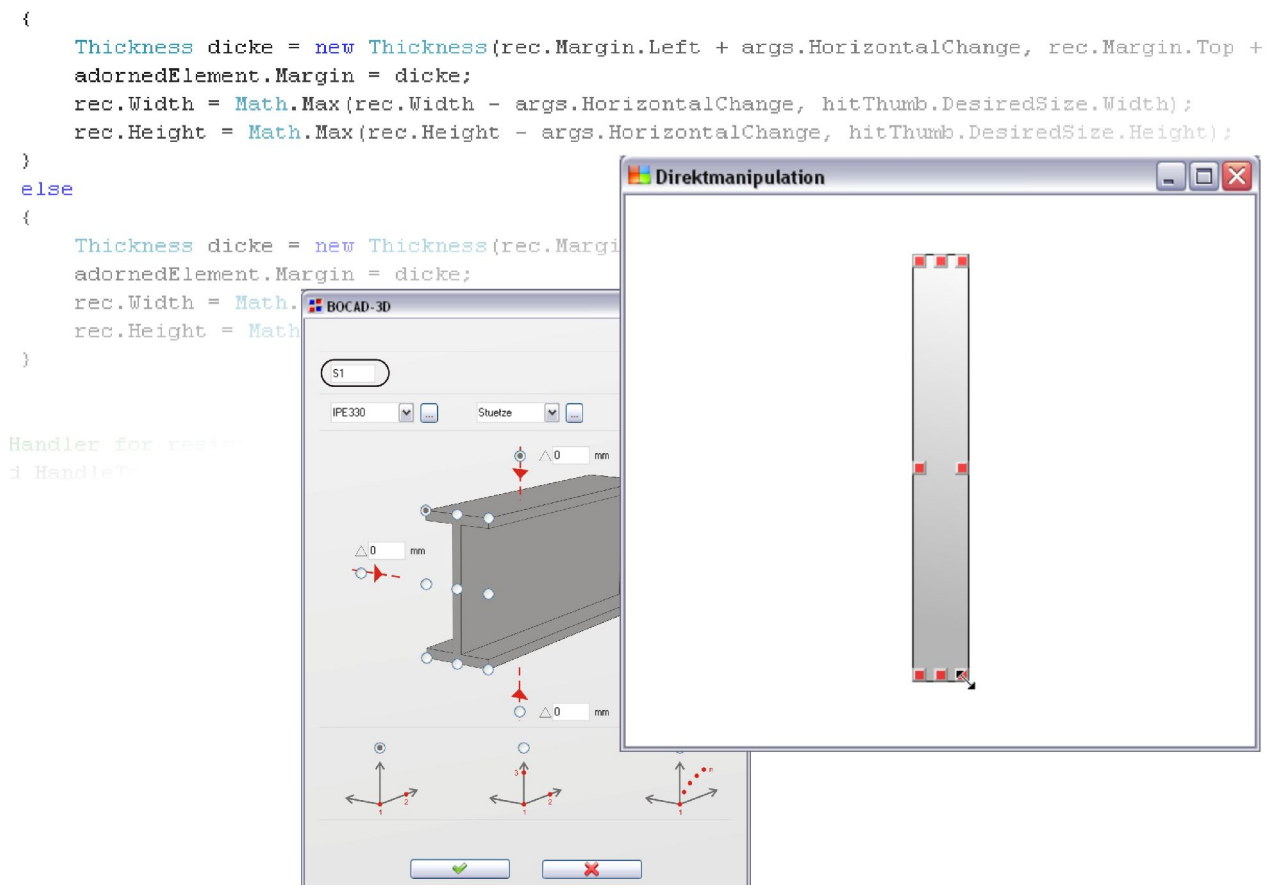
Dipl.-Ing. Daniel Schulten

Wuppertal, Juli 2007

Fachbereich D, Abteilung Bauingenieurwesen
Bergische Universität Wuppertal

Lehr- und Forschungsgebiet für
Theoretische Methoden und Angewandte Informatik

Programmtechnischer Ansatz für Effizienz und Verständlichkeit von Bausoftware in weltweiten Netzen



Dipl.-Ing. Daniel Schulten
Wuppertal, Juli 2007

Diese Dissertation kann wie folgt zitiert werden:

urn:nbn:de:hbz:468-20070716

[<http://nbn-resolving.de/urn/resolver.pl?urn=urn%3Anbn%3Ade%3Ahbz%3A468-20070716>]

Dissertation

Dissertationsschrift eingereicht:

10. April 2007

Mündliche Prüfung und Disputation:

9. Juli 2007

Prüfungskommission:

Univ.-Prof. Dr.-Ing. Georg Pegels (1. Gutachter)

Univ.-Prof. Dr.-Ing. Reinhard Harte (2. Gutachter)

Prof. Dr. Hamid Isfahany

Univ.-Prof. Dr.-Ing. Andreas Schlenkhoff (Vorsitzender)

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehr- und Forschungsgebiet Theoretische Methoden und Angewandte Informatik des Fachbereichs D Abt. Bauingenieurwesen an der Bergischen Universität Wuppertal.

Mein Dank gilt in besonderem Maße Herrn Prof. Dr.-Ing. Georg Pegels für die wissenschaftliche Unterstützung sowie für die vielseitigen Anregungen und Diskussionen während der letzten Jahre, die einen großen Anteil an dem Gelingen dieser Arbeit haben.

Daneben geht mein herzlicher Dank an Herrn Prof. Dr.-Ing. Reinhard Harte für sein Interesse an meiner Arbeit und für die Übernahme des Koreferates.

Ebenso danke ich Herrn Prof. Dr. Hamid Isfahany für seine Mitwirkung als weiteres Mitglied in meiner Prüfungskommission.

Für die Übernahme des Vorsitzes in meinem Prüfungsverfahren und seine hilfreichen Hinweise danke ich Herrn Prof. Dr.-Ing. Andreas Schlenkhoff.

Zusätzlich geht mein Dank an meine Kollegen im Lehrgebiet sowie an Thomas und Julius von netzkern. Julius möchte ich zudem für seine hilfreichen Ratschläge und seine geduldige Hilfsbereitschaft danken.

Mein ganz besonderer Dank gilt Michaela und meinen Eltern für ihren Rückhalt und Ansporn und dass sie mir in allen Phasen meiner Arbeit zur Seite standen.

Wuppertal, im Juli 2007

Daniel Schulten

Zusammenfassung

Zeichnungsbasierte Bausoftware bietet in besonderem Maße die Gelegenheit, intuitiv zu nutzende, weitgehend sprach- und textfreie grafische Benutzungsoberflächen zu gestalten. In diesem Sinne „direkte“ Benutzungsoberflächen vermeiden möglichst die heute typischerweise ineffizienten Umwege über Hauptmenüs, Untermenüs und Listenauswahl oder Symbolanhäufungen am äußeren Bildrand, weit weg vom dargestellten Objekt, das bearbeitet werden soll. Bauzeichnungen und Symbole, die interkulturell verstanden werden, sind ein bisher weitgehend ungenutztes Potential für „intelligente“, ohne Text verständliche Elemente von Benutzungsoberflächen.

Für die Gestaltung von Benutzungsoberflächen gibt es bisher keinen weltweit gültigen Standard. Anhand von marktüblichen, teils vorbildlichen Hersteller-Konventionen und der europäischen Norm 9241 analysiert diese Arbeit deshalb zunächst weltweit eingesetzte Standardsoftware sowie fachspezifische Anwendungen des Bauwesens mit dem Ziel, darauf aufbauend die mit textfreien grafischen Benutzungsoberflächen verbundenen programmtechnischen Fragestellungen zu lösen, die bisher nicht wissenschaftlich untersucht wurden.

Mit diesen Grundlagen leitet die Arbeit einen neuen programmtechnischen Ansatz für die Programmierung von direkt am Bauteil-Objekt sich anbietenden Benutzungsoberflächen her.

Dazu entwickelte Software-Prototypen zeigen als experimenteller Nachweis der Theorie die Entwicklungsschritte und demonstrieren anschaulich, welche neuen Entwicklungsumgebungen und Klassenbibliotheken in diesem Bereich der Bauinformatik grundsätzlich neue Lösungsansätze unterstützen, die dem Re-Engineering weltweit eingesetzter Bausoftware dienen.

Dadurch wird auch den weniger großen Entwicklungsfirmen für Bausoftware der Weg aufgezeigt, wie zukünftig in weltweiten Netzen eingesetzte Bausoftware für den Weltmarkt entscheidend geeigneter entwickelt werden kann, als mit bisherigen Ansätzen möglich. Für das Bauwesen insgesamt erschließen sich dadurch internationale Kooperationsmöglichkeiten im Auslandsbau in einem Maße, wie es bisher zumindest für mittelständische Baufirmen undenkbar war.

Abstract

Drawing based building software offers in particular the opportunity to design intuitively usable widely language- and text-free graphic user interfaces. In these terms, „direct“ user interfaces possibly avoid today’s typical inefficient detours over main menus, sub menus, list selection, or symbol accumulations at the outermost window border, far away from the displayed object which is supposed to be processed. Construction drawings and symbols, which are understood intercultural, are a so far extensively unused potential for „intelligent“ without text understandable elements of user interfaces.

Up to now, a globally valid standard for the design of user interfaces does not exist. With exemplary manufacturer’s conventions and the European Code 9241 this work at first analyses the globally used standard software, as well as subject specific applications of civil engineering with the purpose to solve the thereby program technical questions connected to the user interfaces which until now have not been scientifically tested.

With these basics the work guides a new technical approach for the programming of user interfaces which provides itself directly from the component object.

Developed software prototypes for this display the developmental steps as an experimental proof of the theory and clearly demonstrate which new development surroundings and class libraries in this area of the civil engineering computer science basically support new solution approaches which serve the re-engineering of the globally used building software.

Thereby the way is also shown to the lesser sized development companies for building software how in the future the used civil engineering software in global nets can be developed more conveniently for the global market as it has been possible until now. Therefore, for whole civil engineering international co-operation opportunities in foreign markets develop to an extent which until now has been at least for medium construction companies unimaginable.

Sprachkonventionen, Begriffsdefinitionen

Eine Forschungsarbeit, die sich in deutscher Sprache mit einem Thema beschäftigt, in dem die Wissenschaftssprache Englisch ist, steht zum einen vor der Problematik, die korrekten und etablierten Fachbegriffe zu verwenden, und zum anderen vor der Frage, ob die englischen Originalbegriffe oder deren deutsche Entsprechung verwendet werden soll [Balz00].

Entsprechend dem Standardwerk der Informatik von Helmut Balzert [Balz00] werden in dieser Arbeit die deutschen Begriffe verwendet, sofern sie sinnvoll und üblich sind, z.B. Entwurf anstatt *Design* oder Werkzeug für *Tool*.

Ist aus den deutschen Begriffen nicht unmittelbar erkennbar oder bekannt, welche englische Bedeutung dahinter steht, wird der englische Begriff in Klammern kursiv dahinter gesetzt. Wenn es noch keinen eingebürgerten deutschen Begriff gibt, wird der englische Originalbegriff in kursiver Schreibweise gesetzt um kenntlich zu machen, dass es sich um einen englischen Begriff handelt.

Inhaltsverzeichnis

1	Motiv und Ziel der Arbeit.....	3
2	Stand der Wissenschaft und Technik.....	5
2.1	Historische Entwicklung von Benutzungsoberflächen.....	5
2.2	Konventionen für Benutzungsoberflächen.....	10
2.3	Analyse marktbeherrschender Software und Bausoftware.....	29
3	Konzeption „direkter“, sprach- und textfreier Benutzungsoberflächen.....	62
3.1	Grundlagen.....	62
3.2	Entwurf einer effizienten Benutzungsoberfläche für Bausoftware.....	66
4	Programmtechnischer Ansatz.....	71
4.1	Anforderungsprofil an Programmiersprachen.....	71
4.2	Lösungsweg.....	76
4.3	Grundlagen der objektorientierten Programmierung.....	79
5	Experimenteller Nachweis und Praxiserfahrungen.....	82
5.1	Rahmenbedingungen und eingesetzte Werkzeuge.....	82
5.2	Software-Prototyp I: Menü- und Symbolleisten am Beispiel eines Texteditors.....	83
5.3	Software-Prototyp II: Automatische Übersetzung von unvermeidbaren Texten.....	97
5.4	Software-Prototyp III: Elementares CAD-System mit sprach- und textfreier Benutzungsoberfläche.....	106
5.5	Software-Prototyp IV: Direkte, interaktive Manipulation mittels Anfassern.....	119
6	Ausblick, weitere Forschung.....	142
	Literaturverzeichnis.....	144
	Anhang.....	148

1 Motiv und Ziel der Arbeit

Computer sind in der Welt bis in die Entwicklungsländer hinein milliardenfach verbreitet. Bestenfalls unterstützen sie uns bei der alltäglichen Arbeit, machen diese einfacher und helfen uns, effizienter zu arbeiten.

Dazu muss allerdings die eingesetzte Software durchdacht aufgebaut, fehlerfrei programmiert und mit einer intuitiv bedienbaren Benutzerschnittstelle ausgerüstet sein. Diese hat also entscheidenden Anteil am Erfolg der Software genau an der Schlüsselstelle zum Anwender. Die grafische Benutzungsoberfläche ist also der entscheidende Teil der Software, mit dem der Benutzer im täglichen Gebrauch im Dialog steht und den er als Programm wahrnimmt. Sie ist zudem die einzige Schnittstelle, die sowohl die Anwender des Programms als auch die Entwickler verstehen. Somit stellt sie das ideale Pflichtenheft dar.

Dabei umfasst die grafische Benutzerschnittstelle nicht nur so offensichtliche Elemente wie Symbole, Menüleisten oder Schaltflächen. Sie umfasst auch metaphorische, also adaptierte Verhaltensweisen aus der realen Welt. Ein typisches Beispiel ist „Drag & Drop“, das Ziehen von Objekten und Loslassen, um eine Aktion auszuführen. Das Ziehen einer Datei auf das Papierkorb-Symbol um die Datei zu löschen ist ein typisches Beispiel ebenso das kontextabhängige Verändern des Zeigersymbols, um unterschiedliche Modi anzuzeigen (Doppelpfeil für Größenänderung, gebogener Pfeil für Drehung etc.). Alle Eingabegeräte, z.B. Tastatur und Maus, sind Teil der Benutzerschnittstelle.

Die tägliche Praxis mit marktführender Software zeigt durch massive effizienz hindernde Ereignisse, dass die Gestaltung der Benutzungsoberfläche bei der Entwicklung von Software wissenschaftlich noch unterentwickelt ist. Die Benutzungsoberfläche erscheint zum Schluss „drübergestülpt“ und der Anwender muss sich mit den Vorgaben zurecht finden, die die anwendungsfremden Programmierer entwickelt haben [Rask00].

Die Gestaltung heutiger Software ist geprägt von einigen Global Playern wie Microsoft mit dem „Windows PC“ oder Apple mit dem „Apple Macintosh“. Der Benutzer akzeptiert notgedrungen diese Vorgaben für seine Arbeit und verlangt gleichzeitig aus Gründen der Lerneffizienz, dass auch weitere Softwareprodukte die gleiche, gewohnte Gestaltung aufweisen. So hat jeder Hersteller eigene Konventionen (*GUI-Guidelines*). In ihnen wird

beschrieben, wie Software für die jeweilige Plattform gestaltet werden sollte, für die Windows-Plattform von Microsoft z.B. [MS-01].

In EN ISO 9241 werden nur sehr allgemeine Anforderungen zu Bildschirmarbeitsplätzen aufgestellt, von der Arbeitsumgebung über die Hardware bis hin zur Software. Auch allgemeine Grundsätze zur Dialoggestaltung, Informationsdarstellung und Dialogführung werden genannt [EN-9241].

Darüber hinaus ist kein Standard festgelegt, der für die Internationalisierung von Software relevant wäre. Obwohl der Markt global wird und auch Branchensoftware in alle Länder der Welt verbreitet wird, basieren ihre Benutzungsoberflächen noch immer weitgehend auf Texten, ausgeprägt auch für Bausoftware. Dabei bietet die Kategorie der Bausoftware in besonderem Maß die Chance, einen Standard für eine weltweit verständliche sprach- und textfreie Benutzungsoberfläche zu schaffen. Die standardisierten Bauzeichnungen und Darstellungen bieten ein großes Potential internationale Icons und Symbole zu entwickeln, die es ermöglichen, auf Sprache in der Benutzungsoberfläche weitgehend zu verzichten. Insbesondere die Bauzeichnungen selbst sind international verständlich.

Diese Arbeit leistet daher einen Beitrag zu den Grundlagen für einen Weltstandard für Benutzungsoberflächen von Bausoftware. Erkenntnisse der dritten Phase des DFG-Schwerpunktprogramms 1103 „Vernetzt-kooperative Planungsprozesse im Konstruktiven Ingenieurbau“, speziell der Aspekt allgemeingültiger Benutzungsoberflächen für überwacht koordinierte Planungsprozesse, werden einbezogen. Die laufenden Untersuchungen von Weckmann [Weck07] und Azadnia [Azad07] liefern grundlegende Anregungen und Hinweise direkt aus internationaler Baupraxis im Auslandsbau, aus denen die Konsequenzen für den programmtechnischen Ansatz dieser Arbeit hergeleitet werden.

Der entwickelte programmtechnische Ansatz steigert dabei sowohl die Effizienz des Benutzers bei der Anwendung des Programms als auch die Effizienz des Programmentwicklers durch Verwendung der in dieser Arbeit vorgestellten Klassenbibliotheken.

2 Stand der Wissenschaft und Technik

2.1 Historische Entwicklung von Benutzungsoberflächen

Die Entwicklung von grafischen Benutzungsoberflächen (engl.: Graphical User Interface, GUI) wurde durch die Erfindung der Computermouse möglich, die hardwareseitig die stürmische Entwicklung von Grafikkarten bedingte. Die „Maus“ (siehe Bild 1) gestattet dem Benutzer, durch Handbewegungen mit dem Rechner zu interagieren. Bereits 1963/64 hatten Douglas C. Engelbart und William English am Stanford Research Institute die Computermouse entwickelt. Da es damals allerdings noch keine grafischen Benutzungsoberflächen gab und die Benutzer an Texteingaben gewohnt waren, fand die innovative Entwicklung wenig Beachtung [WIKI-Maus].



Abb. 1: Microsoft Mouse

Quelle: <http://www.winhistory.de/more/pics/mouse.jpg>

Erst 1973 begann die Entwicklung von grafischen Benutzungsoberflächen im Xerox PARC (Palo Alto Research Center) [WIKI-GUI]. Die erste kommerzielle Verwendung war 1983 mit dem Rechner STAR von Xerox. Grundlegende, bis heute gültige Regeln für die Gestaltung von Benutzungsoberflächen stammen aus dieser Zeit [Hell05]. Vor allem die

benutzerorientierte Entwicklung machte diesen Rechner zu einem Meilenstein der Informatik.

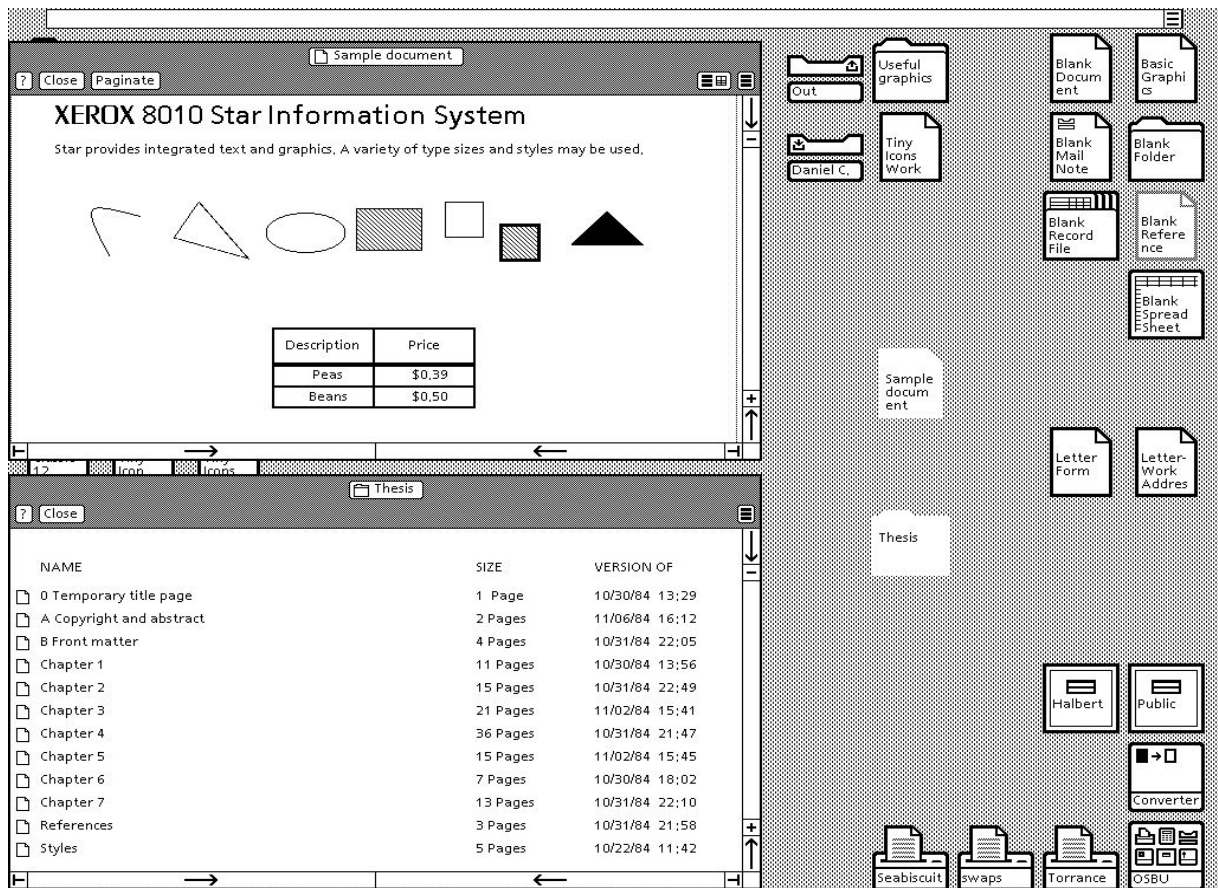


Abb. 2: Benutzungsoberfläche Xerox STAR

Quelle: <http://toastytech.com/guis/starbitmap2.gif>

Die damals neuartige Benutzerfreundlichkeit (Bild 2) beruhte vor allem auf der Umsetzung folgender Prinzipien [Hell05]:

- Der Benutzer arbeitete auf einer virtuellen Schreibtischoberfläche, noch heute als „Desktop“ bezeichnet
- Fenster, Menüs und Scrollbalken erleichterten die Interaktion
- Objekte wurden durch Mausklick auf das Objekt ausgewählt
- Alle Arbeitsvorgänge geschahen durch direkte Manipulation der abgebildeten Objekte: Zeigen und Auswählen. Das war effizienter als Erinnern und Eingeben, wie damals bei kommandogesteuerten Oberflächen erforderlich

- WYSIWYG: What You See Is What You Get. Die Ausgabe auf dem Drucker entsprach der Darstellung auf dem Monitor – ein bis heute gängiger Standard

Erst 1985 gelang Apple mit dem Macintosh (siehe Bild 3) der kommerzielle Durchbruch eines Rechners mit grafischer Benutzeroberfläche [Hell05]. Hauseigene Richtlinien der Firma Apple für die Gestaltung der Benutzeroberflächen sicherten die Konsistenz der Entwicklung über alle Anwendungen, so dass für den Benutzer eine einheitliche Welt entstand. Diese einheitlichen *Guidelines* werden bis heute von den großen Betriebssystemherstellern gepflegt [MS-01], [Appl-01].

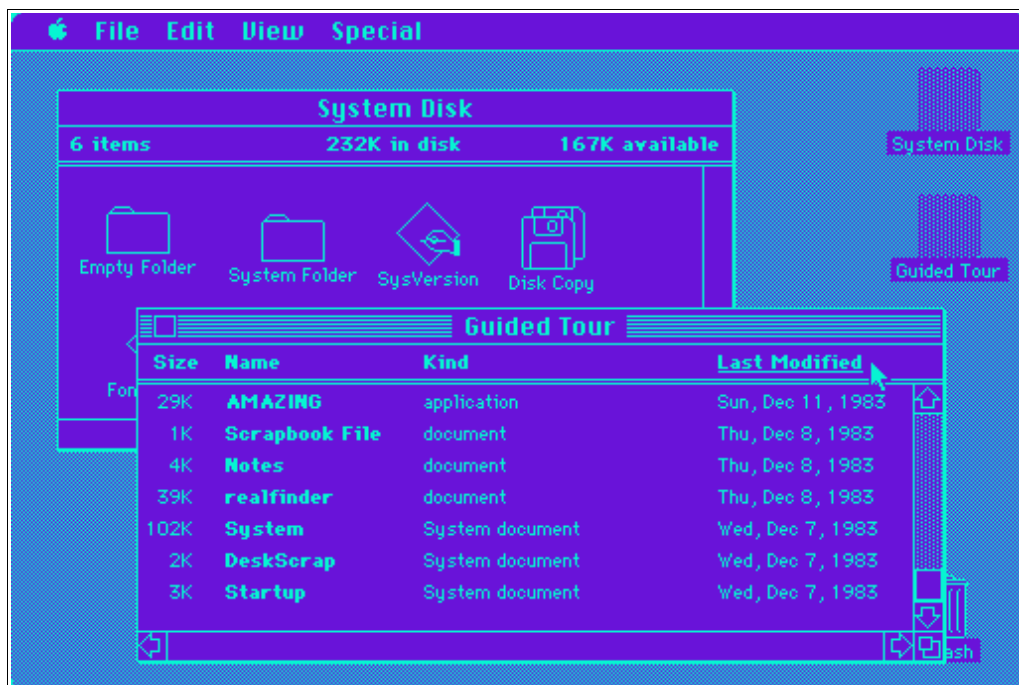


Abb. 3: Screenshot Apple Macintosh, 1985

Quelle: <http://toastytech.com/guis/mac11sortview.gif>

Erst 1990 entwickelte auch Microsoft mit Windows 3.0 ein Betriebssystem mit grafischer Benutzeroberfläche – nach dem Vorbild des Macintosh.

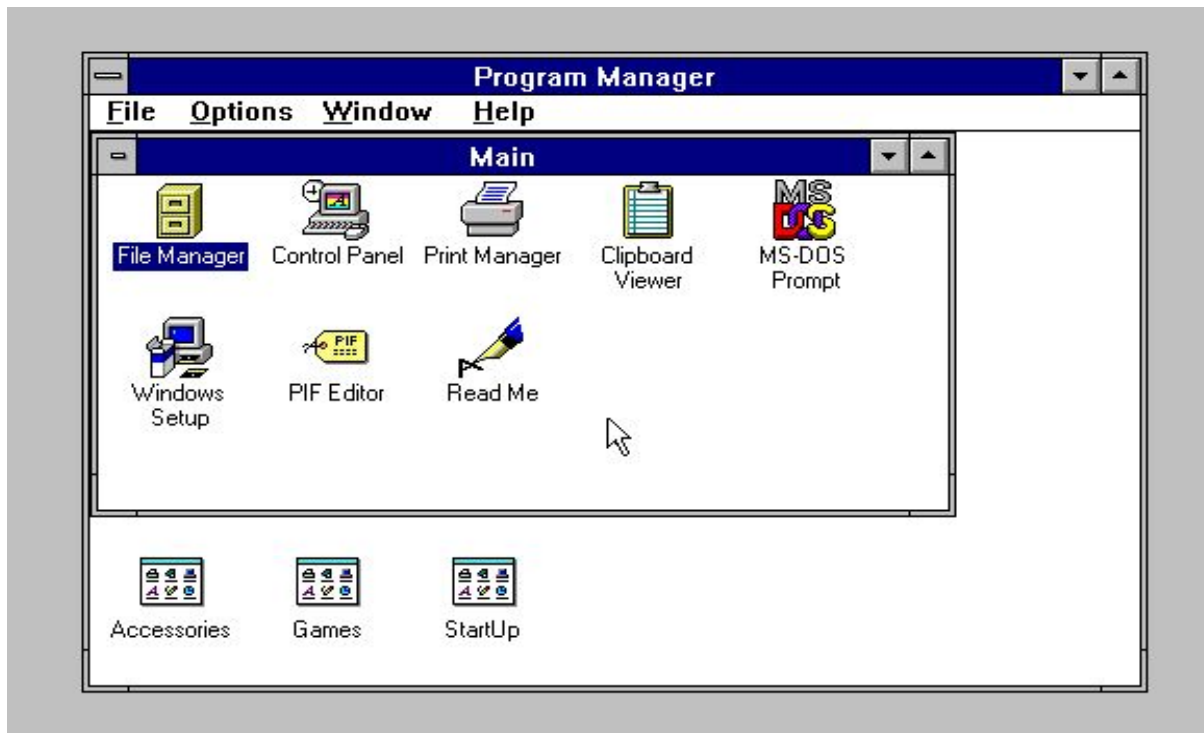


Abb. 4: Screenshot Microsoft Windows 3.1

Quelle: <http://www.aresluna.org/guidebook/screenshots/win31>

Diese Version verfügt über alle noch heute üblichen Merkmale einer grafischen Benutzeroberfläche: Der Benutzer arbeitet mit der Maus auf der virtuellen Darstellung eines Schreibtisches, Symbole stellen metaphorisch Elemente aus dem realen Leben dar (z.B. Drucker, Aktenschränke etc.), die der Benutzer mit der Maus anklicken kann (siehe Bild 4). So beginnt mit Windows 3.0 die Etablierung eines de-facto Standards.

Gleichzeitig war eine stürmische Entwicklung der Systemhardware zu beobachten, die grafische Leistungen immer stärker und konsequenter unterstützte und damit wiederum neue Formen von grafischen Benutzeroberflächen ermöglichte. So hat die dadurch mögliche, massenhafte Verbreitung von Computerspielen die technische Entwicklung und die wissenschaftliche Forschung zu grafischer Datenverarbeitung ungemein befruchtet. Als Folge sind die heutigen Systeme farbenreich und animiert. Objekte, die mittels „Drag & Drop“, dem Ziehen und Loslassen mit der linken Maustaste, bewegt werden, erscheinen transparent. Beim Leeren des Papierkorbes ertönt z.B. ein typisches Geräusch und das Papierkorbsymbol ändert sich von einem mit zerknittertem Papier gefüllten Korb in einen leeren. Was zum Teil als Spielerei anmutet, kann zukünftig bei gut

durchdachtem Einsatz für Bausoftware die Bedienungseffizienz und Verständlichkeit massiv verbessern, wie noch gezeigt wird.

Auch die Opensource-Betriebssysteme wie Linux übernehmen diesen Trend und imitieren im Aussehen und Verhalten die Betriebssysteme der Marktführer. Zur Lerneffizienz wird Linux wahlweise mit Stilen ähnlich denen von Windows oder Apple ausgeliefert. Somit kann die spezifische Betrachtung von Linux in dieser Arbeit entfallen.

Unix ist bis heute ein professionelles Betriebssystem für Wissenschaftler mit geringen Marktanteilen geblieben. Aufgrund dieser Nischenbeschränkung wird das Betriebssystem Unix in dieser Arbeit nicht weiter betrachtet.

Screenshots der im Jahre 2006 aktuellen Betriebssysteme Microsoft Windows XP und Apple OS X zeigen die folgenden Bilder 5 und 6. Im Rahmen dieser Arbeit ist nun von Interesse, wie derartige grafische Oberflächen programmiert werden und vor allem, ob und wie die Oberfläche von Bausoftware von den relativ kleinen Entwicklungsfirmen ebenso programmiert werden kann. Der wohl überlegte Einsatz von grafischen Leistungselementen dieser Betriebssysteme für Bausoftware lässt eine erhebliche Effizienzsteigerung der Bedienung erwarten.

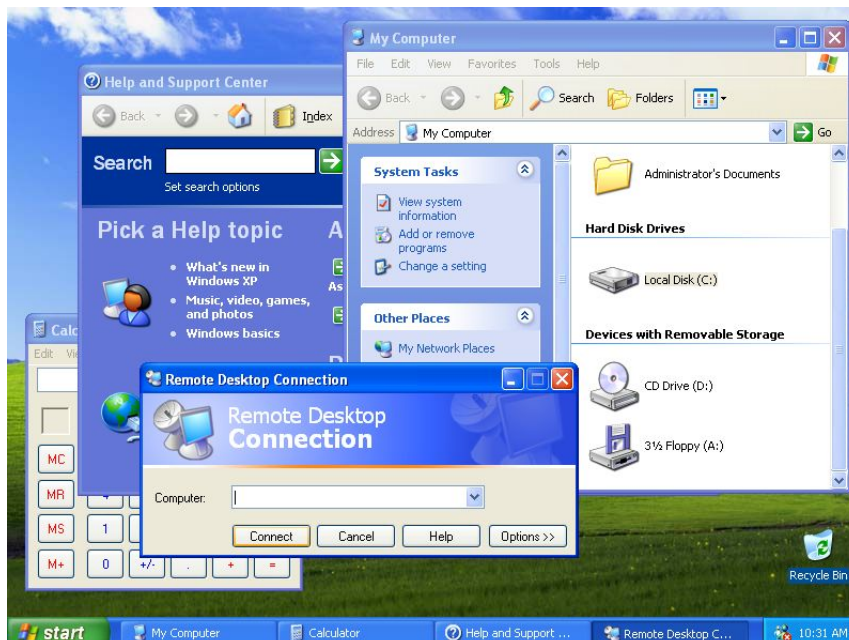


Abb. 5: Screenshot Windows XP

Quelle: <http://www.aresluna.org/guidebook/screenshots/winxppro>



Abb. 6: Screenshot Apple OS X Panther

Quelle: <http://www.aresluna.org/guidebook/screenshots/macosex103>

2.2 Konventionen für Benutzungsoberflächen

Marktprägend für Benutzungsoberflächen sind wie bereits erläutert die Guidelines von Microsoft und Apple, deren Erscheinungsbild von freien Betriebssystemen wie z.B. Linux nachempfunden wird, um so von deren Marktdominanz zu profitieren und Benutzern den Umstieg zu erleichtern.

Zur Erläuterung der in dieser Arbeit besonders interessierenden Entwicklungssoftware für Benutzungsoberflächen wird im späteren Verlauf dieses Kapitels auf Apples aktuelle API Cocoa sowie Microsofts Gegenstück Luna und ihre Guidelines eingegangen.

API ist dabei die Abkürzung für Application Programming Interface. Eine API ist die Schnittstelle, die ein Betriebssystem anderen Anwendungsprogrammen z.B. der Bau- software zur Verfügung stellt. Sie enthält also Routinen, Protokolle und Dienstprogramme für das Erstellen von Software durch Dritte [WIKI-API].

Grundsätzlich dienen APIs der Einheitlichkeit von Anwendungssoftware auf einem System. So existieren neben APIs für den Zugriff auf die Hardware für Hersteller von Fest-

platten oder Grafikkarten auch APIs für die Erstellung der grafischen Benutzungsoberflächen für Anwendungen. Nutzen alle Programmierentwickler dasselbe API für das Erstellen, Zeichnen und Verwalten von Fenstern einer Benutzungsoberfläche, sehen alle Fenster gleichartig aus und besitzen dieselben Komponenten an denselben Stellen. Die Arbeit für den Benutzer wird erleichtert, da er die „Bedienung“ des Systems nur einmal erlernen muss.

2.2.1 Normung zu Benutzungsoberflächen, DIN EN-ISO 9241

Die EN-ISO 9241 aus den Jahren 1997 bis 2001 besitzt keinen verbindlichen Charakter [EN-9241]. Sie legt die ergonomischen Anforderungen für Bürotätigkeiten mit Bildschirmgeräten fest um „die Entwickler und Hersteller bei der Entwicklung ergonomisch richtig gestalteter Bildschirmgeräte und Softwaresysteme zu unterstützen“ [EN-9241].

Die Norm soll unter Berücksichtigung der Gestaltungsaufgabe und des Benutzerkontextes der Software verwendet werden, z.B. bei Gestaltung eines Dialoges oder bei Evaluation eines Softwareproduktes.

Es wird explizit darauf hingewiesen, dass sie keinen *Styleguide* (synonym verwendetes Wort für *Guidelines*) für Benutzungsoberflächen darstellt, jedoch bei der Entwicklung von plattformabhängigen Styleguides hilfreich ist. Auf diese Weise sollte durch die Norm die stürmische Entwicklung nicht behindert, sondern gefördert werden.

Von den 17 Teilen der Norm decken die Teile 9241-10 bis 9241-17 Themen zur Software-Ergonomie ab, so z.B. Grundsätze für „Mensch-Computer-Dialoge“ in Teil 10, Definition von Gebrauchstauglichkeit in Teil 11 oder Empfehlungen zur Informationsdarstellung (Teil 12), Dialogführung (Teil 13) und Dialogtechniken (Teil 17).

Die Grundsätze der Norm sind aufgrund der allgemeinen Aussagen auch heute noch gültig. Demgegenüber definieren die firmenspezifischen Styleguides und ihre APIs im Detail den neuesten Stand der Technik. Die relevanten Teile der Norm, die wegen ihrer bleibenden Allgemeingültigkeit auch Basis dieser Arbeit sind, werden nachfolgend diskutiert.

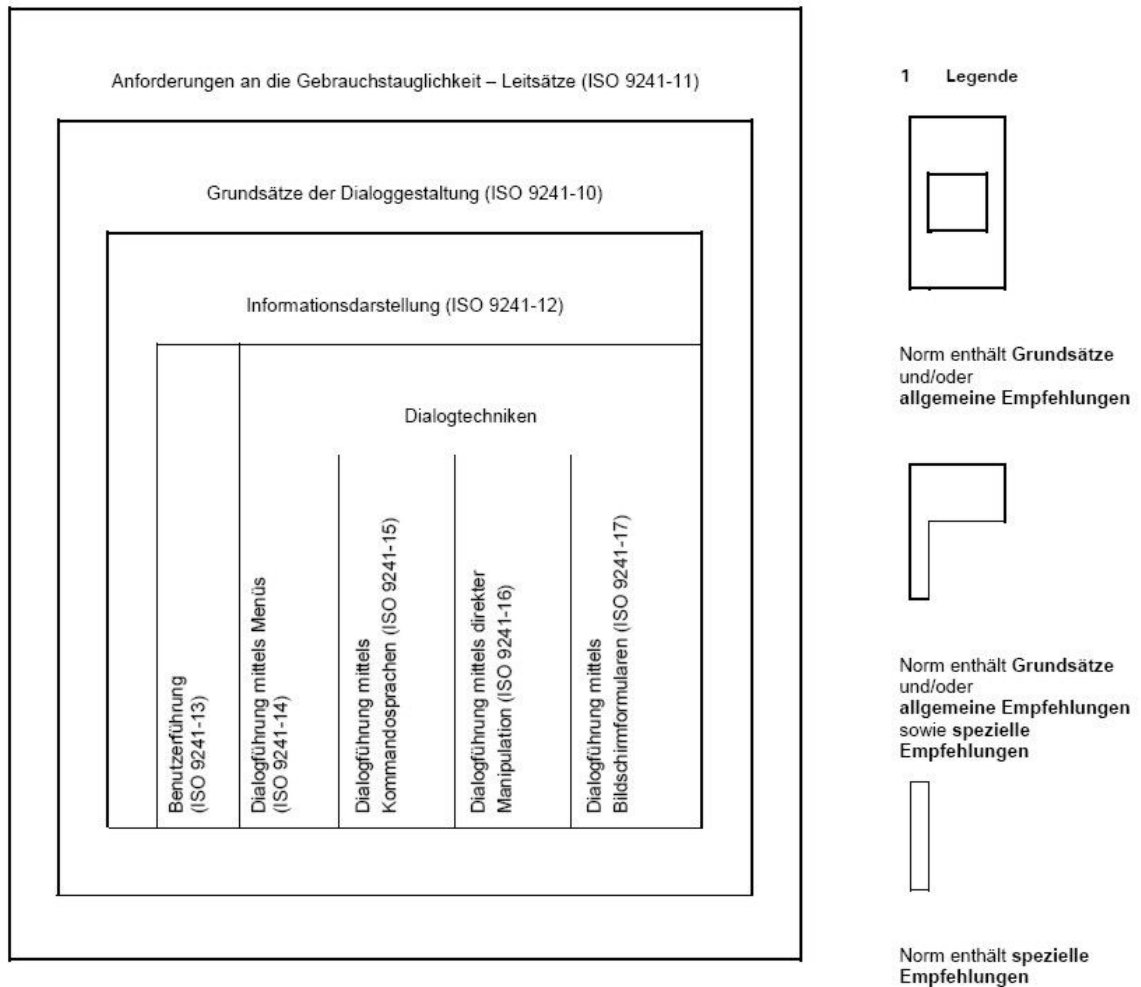


Abb. 7: Aufbau der Norm ISO 9241, Quelle: EN-ISO 9241-1, Seite 12

9241-10 „Grundsätze der Dialoggestaltung“

Entsprechend Abb. 7 definiert Teil 10 der Norm die übergeordneten ergonomischen Grundsätze, die für grafische Benutzungsoberflächen anzuwenden sind. Er definiert sieben Grundsätze (lt. [EN-9241]) für die Gestaltung des Dialogs zwischen Benutzer und Programm:

– Aufgabenangemessenheit

Ein Dialog ist aufgabenangemessen, wenn er den Benutzer unterstützt, seine Arbeitsaufgabe effektiv und effizient zu erledigen.

- Selbstbeschreibungsfähigkeit
Ein Dialog ist selbstbeschreibungsfähig, wenn jeder einzelne Dialogschritt durch Rückmeldung des Dialogsystems unmittelbar verständlich ist oder dem Benutzer auf Anfrage erklärt wird.
- Steuerbarkeit
Ein Dialog ist steuerbar, wenn der Benutzer in der Lage ist, den Dialogablauf zu starten sowie seine Richtung und Geschwindigkeit zu beeinflussen, bis das Ziel erreicht ist.
- Erwartungskonformität
Ein Dialog ist erwartungskonform, wenn er konsistent ist und den Merkmalen des Benutzers entspricht, z.B. seinen Kenntnissen aus dem Arbeitsgebiet, seiner Ausbildung und seiner Erfahrung sowie den allgemein anerkannten Konventionen.
- Fehlertoleranz
Ein Dialog ist fehlertolerant, wenn das beabsichtigte Arbeitsergebnis trotz erkennbar fehlerhafter Eingaben entweder mit keinem oder mit minimalem Korrekturaufwand seitens des Benutzers erreicht werden kann.
- Individualisierbarkeit
Ein Dialog ist individualisierbar, wenn das Dialogsystem Anpassungen an die Erfordernisse der Arbeitsaufgabe sowie an die individuellen Fähigkeiten und Vorlieben des Benutzers zulässt.
- Lernförderlichkeit
Ein Dialog ist lernförderlich, wenn er den Benutzer beim Erlernen des Dialogsystems unterstützt und anleitet.

Diese Grundsätze bilden die Grundlage für alle weiteren software-ergonomischen Empfehlungen der ISO 9241.

Die ISO 9241-10 definiert für jeden der genannten Punkte Empfehlungen und gibt entsprechende Beispiele.

Ein typisches und gängiges Beispiel aus dem Bereich der Erwartungskonformität ist die Empfehlung, dass „Änderungen des Dialogzustandes auf einheitliche Art und Weise herbeigeführt werden [sollen]“. Als Beispiel dazu gibt die Norm die F1-Taste an, die durchgängig für den Hilfe-Aufruf verwendet wird.

9241-11 „Anforderungen an die Gebrauchstauglichkeit“

Dieser Teil gibt Merkmale an, die Gebrauchstauglichkeit von Softwareprodukten zu evaluieren. Dazu zählen Effektivität, Effizienz und das zufriedenstellende Erreichen von bestimmten Zielen („Zufriedenstellung“) in einem gegebenen Nutzungskontext. Er beinhaltet vor allem Bewertungs- und Messkriterien, um die Gebrauchstauglichkeit zu bewerten. Diese werden speziell im experimentellen Teil dieser Arbeit herangezogen.

9241-12 „Informationsdarstellung“

Teil 12 der Norm gibt ergonomische Empfehlungen für die Informationsdarstellung für „zeichenorientierte oder grafische Mensch-Maschine-Schnittstellen“ [EN-9241] um visuelle Information für den Benutzer möglichst effektiv, effizient und zufrieden stellend zu präsentieren.

Dafür gibt die ISO 9241-12 folgende Ziele an:

- Klarheit, der Informationsgehalt wird schnell und genau vermittelt
- Unterscheidbarkeit, die angezeigte Information kann genau unterschieden werden
- Kompaktheit, den Benutzern wird nur jene Information gegeben, die für das Erledigen der Aufgabe notwendig ist
- Konsistenz, gleiche Information wird innerhalb der Anwendung entsprechend den Erwartungen des Benutzers stets auf die gleiche Art dargestellt
- Erkennbarkeit, die Aufmerksamkeit des Benutzers wird zur benötigten Information gelenkt
- Lesbarkeit, die Information ist leicht zu lesen
- Verständlichkeit, die Bedeutung ist leicht verständlich, eindeutig interpretierbar und erkennbar

In Abschnitt 5.9.1 gibt die ISO 9241-12 eine wesentliche Vorgabe zur Beschriftung von Bildelementen: So sollten „Bildelemente (z.B. Felder, Elemente, Bildelemente und Grafiken) beschriftet sein, außer wenn ihre Bedeutung offensichtlich und für die vorgesehenen Benutzer klar verständlich ist.“

Sollte eine Beschriftung nicht praktikabel sein, z.B. weil nur begrenzt Raum verfügbar ist, ist eine Schnellinformation („Balloon help“, heutzutage besser bekannt als „Quickinfo“) eine annehmbare Alternative.

9241-13 „Benutzerführung“

In ISO 9241 Teil 13 werden Aspekte der Benutzerführung von Software-Benutzerschnittstellen zusammengestellt:

- allgemeine Empfehlungen zur Benutzerführung
Hierbei werden speziell Hinweise zur Formulierung der Benutzerführung gegeben. So soll u.a. die Benutzerführung grammatikalisch einheitlich formuliert werden sowie Fehlermeldungen spezifisch und nicht zu allgemein formuliert sein.
- Eingabeaufforderungen
Da textförmige Eingabeaufforderungen nicht zu den in dieser Arbeit betrachteten Benutzungsoberflächen zählen, werden diese Empfehlungen der ISO nicht weiter erörtert.
- Rückmeldungen
Mittels Rückmeldungen signalisiert das Softwaresystem dem Benutzer erfolgreiche Eingaben sowie den Systemzustand. Die ISO erläutert einige Empfehlungen zu Rückmeldungen, so z.B. dass Rückmeldungen im normalen Arbeitsablauf unaufdringlich und eindeutig sein sowie zeitnah erfolgen sollen.
- Statusinformation
Statusinformationen sind laut ISO Anzeigen über den aktuellen Zustand der Hard- oder Software, so z.B. über Modi, Prozesse oder verfügbare Anwendungen. Die ISO gibt anhand einer Liste an Bedingungen Empfehlungen, wann ständige, automatische oder vom Benutzer anzufordernde Statusinformationen geeignet sind.
- Fehlermanagement
Unter Fehlermanagement versteht die ISO Fehler bei der „Mensch-Computer-Interaktion“. Fehler sollen vermieden werden, z.B. durch ähnliche Funktionen in verschiedenen Modi, durch das System korrigiert werden oder durch den Benutzer behoben werden können. Gängigstes Beispiel dafür ist die „Undo“-Funktion. Ebenso gibt die ISO Empfehlungen zur Anzeige und Formulierung von Fehlermeldungen.
- Online-Hilfe
Online-Hilfe stellt dem Benutzer laut ISO während der Softwarebenutzung zusätzliche Hilfen zur Verfügung. Dies können systeminitiierte oder benutzerinitiierte Hilfen sein. Für beides gibt die ISO Entscheidungshilfen für die Wahl der Hilfe. Ebenso gibt sie Anregungen zur Darstellung der Hilfe-Information, sowie zur Steuerung und Suche bei kontextfreier und kontextsensitiver Hilfe.

9241-14 „Dialogführung mittels Menüs“

Dieser Teil der ISO enthält Empfehlungen für Menüs und deren Anwendung in Benutzerschnittstellen, z.B. für Fenster, Anzeigebereiche, Schaltflächen und Felder. Die ISO definiert drei wesentliche Entwurfskomponenten: Den Dialog, die Ein- und die Ausgabe.

Dabei ist die Eingabe die Art und Weise, mit der Informationen eingegeben werden können, also z.B. mittels Funktionstasten, Cursorstasten, „Zeigergeräten“ und Spracheingabe. ISO Teil 14 gibt Empfehlungen zum Einsatz eines jeden Eingabemittels.

Der Dialog ist der Teil des Systems, in dem der Benutzer seine Eingaben tätigt, angeleitet durch das System. Dialoge sollen so entworfen werden, dass sie den Benutzer bei der Erledigung seiner Aufgabe unterstützen und nicht durch Besonderheiten des Softwaresystems ablenken.

Die Ausgabe behandelt die konsistente Darstellung der Daten auf dem Bildschirm, d.h. die Anordnung von Menüs, Struktur und Syntax von Textinformationen sowie Darstellungsmöglichkeiten zur Unterscheidbarkeit von Optionen.

Die in der Norm verwendeten Begriffe Ein- und Ausgabe wirken missverständlich. Bessere Begriffe stattdessen wären Ein- und Ausgabewerkzeuge, da die Begriffe Ein- und Ausgabe bereits mit dem Dialog verbunden werden. Ohne Ein- und Ausgabe ist kein Dialog möglich, sie sind also bereits Teil des Dialogs.

9241-16 „Dialogführung mittels direkter Manipulation“

Unter direkter Manipulation wird die direkte Bearbeitung der auf dem Bildschirm angezeigten Objekte verstanden. Dies geschieht über metaphorische Darstellungen, also bildliche Übertragungen aus der realen Welt. So stellt ein Blatt Papier ein Dokument dar, ein Stift steht stellvertretend für die Möglichkeit, Linien zu zeichnen. Auch Tasten und Schieberegler können verwendet werden, z.B. um die Möglichkeit der Lautstärkeanpassung zu geben.

Zur direkten Manipulation zählt die ISO auch gängige Vorgehensweisen wie z.B. „Drag & Drop“ oder die Cursorveränderung bei unterschiedlichen Benutzeraktionen, die bereits Einzug in die etablierten Werkzeuge des modernen Benutzungsschnittstellen-Designs gefunden haben und auch in dieser Arbeit verwendet werden.

Die Konzeption eines für Bausoftware-Entwicklungsfirmen gangbaren Lösungswegs für die Programmierung direkter Manipulation ist der wesentliche Kern dieser Arbeit.

9241-17 „Dialogführung mittels Bildschirmformularen“

Ähnlich wie bei der Dialogführung mittels Menüs enthält Teil 17 Empfehlungen zur Gestaltung der Benutzungsschnittstelle mittels Bildschirmformularen. Hierbei handelt es sich um Dialogfenster, die ähnlich einem Formular am Bildschirm vom Benutzer ausgefüllt werden.

Zumindest textarme Bildschirmformulare, z.B. zur Spezifikation von Zukaufteilen, Dichtungen, etc. werden sich für Bausoftware nicht völlig vermeiden lassen. Daher werden im experimentellen Nachweis dieser Arbeit auch für Bildschirmformulare geeignete programmtechnische Lösungen entwickelt.

2.2.2 Guidelines

Unter dem Begriff „Guidelines“ geben die Hersteller von Betriebssystemen Designrichtlinien heraus, wie Anwendungsprogramme Dritter unter ihrem Betriebssystem gestaltet werden sollen.

Als maßgebend gelten hierbei weltweit die Konventionen von Microsoft und Apple für die Betriebssysteme Windows und Apple OS X.

Beide Hersteller veröffentlichen Ihre Guidelines im Internet, damit sie stets auf dem aktuellsten Stand der Technik verfügbar sind und von Dritten ohne Aufwand eingesehen werden können.

Bei näherer Analyse fällt auf, dass die Richtlinien von Apple die weitaus detailliertesten und hilfreichsten sind. Im Gegensatz dazu fallen die Richtlinien von Microsoft durch mangelnde Übersichtlichkeit, schlechte Navigation durch die Website und eine veraltete API-Beschreibung auf.

Aus diesem Grunde werden die richtungweisenden Designrichtlinien von Apple vor denen von Microsoft vorgestellt. Die Verbreitung von Apple-Computern im Bauwesen ist zwar nur in Architekturbüros hoch. Die innovativen Ansätze von Apple werden aber von

allen Anbietern nachgeahmt. Auf diesem Wege finden sich so später viele Grundsätze der Gestaltung von Apple-Benutzungsoberflächen auch in Microsoft-Produkten.

Die programmtechnische entsprechende Konsequenz prägender Guidelines speziell für Bausoftware zu analysieren und Lösungswege aufzuzeigen, ist Hauptteil dieser Arbeit.

„Apple Human Interface Guidelines“

Der Apple Styleguide [Appl-01] beschreibt nicht nur die Gestaltungsrichtlinien für Apples aktuelle Grafik-API „Cocoa“ mit der Oberfläche „Aqua“, sondern gibt dem Gestalter von Benutzungsoberflächen zusätzlich ein umfassendes Kompendium und Nachschlagewerk. Das Themenspektrum ist dabei so umfassend, dass es die Gestaltung von grafischen Benutzungsoberflächen hinreichend abdeckt.

Apple verwendet als lehrreiche Beispiele nämlich Anwendungen, die ein Benutzer täglich braucht und daher gut kennt. Diese sind durchgängig und einheitlich gestaltet, damit nicht nur der Programmentwickler, sondern letztendlich der Benutzer effektiv arbeiten kann.

Aufgrund der Zusammenarbeit von Apple und Intel ab Ende 2005 erhalten die Apple Guidelines weiteres Gewicht. Es gibt Grund zu der Annahme, dass nunmehr Windows-Programme direkt, d.h. ohne Emulation, auf Apple-Rechnern ausgeführt werden können. Auch die neuen sog. „Universal-Programme“ von Apple, die für den Intel-Prozessor geschrieben werden, sind dann auch auf den Windows-PCs ausführbar.

So wird vorstellbar, Windows- und Apple-Programme trotz der Konkurrenz der Hersteller auf ein und demselben System einzusetzen. Eine erste Beta-Version von „Bootcamp“, herausgegeben Mitte 2006 von Apple, lässt bereits das parallele Installieren von Mac OS X und Windows XP auf Apple-Rechnern zu. Die Endversion von Bootcamp soll Bestandteil des neuen Apple-Betriebssystems werden.

In den Apple-Guidelines werden im ersten Kapitel die Grundsätze der Gestaltung von Benutzungsoberflächen erläutert, also grundlegende Designprinzipien, die es bei der Entwicklung zu beachten gilt und die betriebssystemübergreifend auch auf Windows-Applikationen übertragen werden können.

Im zweiten Kapitel wird das sog. „Macintosh Experience“ [Appl-01] vorgestellt. Hier werden Mac OS X spezifische Techniken aufgeführt, an die der Benutzer von Apple-Com-

putern gewöhnt ist. Auch dieser Teil kann durchaus als Anregung für andere Betriebssysteme dienen.

Das spezielle Augenmerk bei der Betrachtung der Apple Guidelines im Rahmen dieser Arbeit liegt allerdings auf dem dritten Kapitel. Hier wird die anzustrebende Gestaltung der Programme Dritter im Rahmen des „Aqua Interface“ detailliert beschrieben.

Im ersten Schritt der Guidelines werden die Werkzeuge der Benutzereingabe definiert. Dazu zählen auf Seiten der Hardware die Maus, das Grafiktablett, das Touchpad und die Tastatur sowie die dazugehörigen Aktionen.

So wird z.B. der „Doppelklick“ definiert als das schnelle, zweifache Klicken mit der linken Maustaste auf ein Objekt. Dadurch wird die objektspezifische Standardaktion ausgeführt, z.B. beim Doppelklick auf eine Datei wird diese geöffnet. Da nicht jeder Anwender in der Lage ist, das zweifache schnelle Klicken korrekt auszuführen, ist der Doppelklick nicht der einzige Weg, diese Aktion auszuführen.

Auch die spezifischen Bedeutungen der Standardtasten auf der Tastatur werden mit der dann vom Benutzer erwarteten Aktion erläutert, so z.B. das Drücken der Eingabetaste. Dies hat bei Textfeldern das Bestätigen der Eingabe zur Folge, in Dialogfenstern ist es identisch mit dem Drücken der Standard-Schaltfläche und in einem Textverarbeitungsprogramm fügt es einen Zeilenumbruch ein.

Ebenfalls gibt Apple vor, welche Tastaturkürzel (*Shortcuts*) mit welcher Bedeutung zu versehen sind. Diese Kürzel (auch *Accelerator* = Beschleuniger) sind Abkürzungen für besonders häufig verwendete Aktionen. Diese sollen programmübergreifend dieselbe Aktion ausführen. Apple gibt in seinen Guidelines vor, welcher Shortcut welche Aktion ausführt: So soll das Drücken der Taste „Esc“ den aktuellen Prozess abbrechen. Ebenfalls gut bekannt ist „Strg+C“ und „Strg+V“ zum Kopieren und Einfügen von Objekten (z.B. Texten).

Programmentwickler sollen für die Aktion „Drücken der Eingabetaste“ stets die in diesem Zusammenhang übliche Reaktion vorsehen. Dies ist konkret die Forderung der [EN-9241] nach „Erwartungskonformität“: Der Benutzer soll erhalten, was er erwartet. Mit derartigen Vorgaben legt Apple fest, wie ein Softwareprodukt auf Benutzereingaben zu reagieren hat, damit sie dem Benutzer in „Fleisch und Blut“ übergehen.

Wesentliches Merkmal einer konsistenten Benutzungsoberfläche sind auch identische Methoden zur Auswahl von Objekten. Apple nennt die Implementierung der Drag & Drop-Funktion als wesentlichen Grund für die einfache und intuitive Benutzung ihres Betriebssystems und legt daher großen Wert auf die korrekte und systemkonsistente Umsetzung dieser Funktion, auch in Programmen Dritter.

Die Implementierung von Drag and Drop soll dabei zielspezifisch „intelligent“ sein: Wird ein Adressbucheintrag in die Adresszeile einer e-Mail kopiert, soll die e-Mail Adresse eingefügt werden und nicht die kompletten Adresdaten. Zudem sollen sich Drag and Drop Aktionen rückgängig machen lassen.

Die Semantik von Drag and Drop legt fest, dass das Ziehen eines Objektes im selben Dokument das Objekt verschiebt, beim Ziehen in ein anderes Dokument wird es kopiert. Dies ist dem Benutzer durch eine Änderung des Mausursors zudem anzuzeigen, z.B. durch das „+“-Symbol.

Auch bezüglich der Darstellung von Texten, die sich auch im Ansatz dieser Arbeit nicht gänzlich vermeiden lassen, gibt der Styleguide Vorgaben. So soll, unabhängig von eigenen Vorlieben des Entwicklers, die auch für ältere Anwender gut lesbare Systemschrift „Lucida“ in verschiedenen Größen und Schriftschnitten verwendet werden. Diesen Systemfont und Minisystemfont zeigt Bild 8.

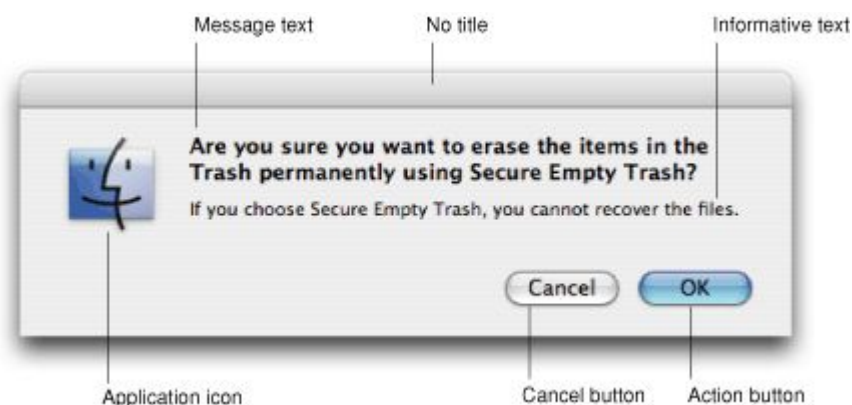


Abb. 8: MAC OS X Dialogfenster

Quelle: [Appl-01]

MAC OS X unterstützt den Entwickler hier, in dem es vordefinierte Funktionen anbietet. So erhält ein Text über den Eigenschaftswert `kThemeSystemFont` den Systemfont für

seine Darstellung auf dem Bildschirm bzw. über `kThemeMiniSystemFont` den als `Minisystemfont`.

Neben der Einheitlichkeit in Anwendungen hat die Verwendung dieser Systemfunktionen einen weiteren Vorteil: Passt der Benutzer die Darstellungsoptionen seinen persönlichen Bedürfnissen an, z.B. durch die Verwendung von extragroßer Schrift oder der Änderung der Farben, wird auch jede Anwendung mit diesen Einstellungen dargestellt.

Icons, den grafischen Symbolen, ist ein eigener Abschnitt im Apple-Styleguide gewidmet. Neben der Klassifizierung in z.B. Anwendungs-, Werkzeug- und Toolbar-Icons wird erläutert, wie Icons unter MAC OS zu gestalten sind.

So wird z.B. für Toolbar-Icons (entspricht der Taskleiste in Windows) die Darstellungssicht „von vorn“ festgelegt, siehe Abb. 9. Das Material des Objektes ist realistisch darzustellen. Transparenz soll beispielsweise nur dann genutzt werden, wenn es dem Material entspricht, z.B. bei einem Vergrößerungsglas.



Abb. 9: Kameraperspektive zur Gestaltung von Toolbar-Icons
Quelle: [Appl-01]

Apple verfolgt hier also einen realitätsnahen Ansatz, welcher mit der Forderung der EN-ISO nach metaphorischer Darstellung in Einklang steht. Der Benutzer soll das Gefühl haben, Objekte aus der Realität zu nutzen, deren Sinn sich für ihn schnell und intuitiv erschließt.

Die jeweilige Verwendung der verschiedenen Mauszeiger ist vorgeschrieben und durch entsprechende Funktionen der API definiert. Der Entwickler kann auf diese Funktion an entsprechender Stelle zugreifen und so in seiner Anwendung die Cursorform ändern, um z.B. mit dem Symbol $\leftarrow \rightarrow$ das Verschieben einer Linie anzuzeigen.

Ein großes Kapitel ist der Gestaltung von Menüs gewidmet. Dabei werden Hauptmenüs, Kontextmenüs und die für Apple typischen „Dockmenüs“ behandelt. Dockmenüs haben

keine hinreichend passende Entsprechung bei Windows und sind somit weltweit nicht hinreichend bekannt. Daher werden sie in dieser Arbeit nicht verwendet.

Der Styleguide behandelt den Stil und die grafische Gestaltung von Menüs, ohne auf die dahinterstehende Programmierung durch Dritte eingehen zu können.

Dennoch werden sehr sinnvolle Hinweise zur klaren Namensgebung von Menüeinträgen gegeben. So soll z.B. der Menüeintrag „Schrift“ wirklich nur Schriftfamilien zur Auswahl haben, aber keine Befehle zum Bearbeiten von Text wie Ausschneiden und Einfügen.

Menüeinträge werden von Verben, Objekten oder Attributen abgeleitet, z.B. „Speichern“ oder „Kursiv“.

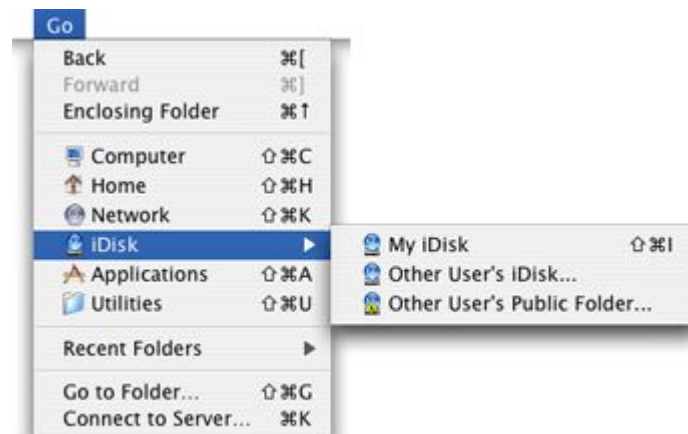


Abb. 10: Menüelemente bei MAC OS X

Quelle: [Appl-01]

Im Gegensatz zum hier verfolgten Ansatz einer möglichst sprach- und textfreien Benutzungsoberfläche gibt der Styleguide Apple explizit an, dass Menüeinträge in Form von Texteinträgen erfolgen sollen. Aber immerhin wird empfohlen, diese zusätzlich durch Icons zu ergänzen, und zwar so, dass sie dann in Symbol- und Werkzeugleisten wiederverwendet werden. Dies entspricht dem in EN-ISO verfolgten Ansatz der Lernförderlichkeit. Nutzt der Benutzer eine neue Funktion, wählt er diese zunächst aus dem textförmigen Menü aus. Dabei prägt er sich automatisch das damit verbundene Icon ein, welches er dann in einer Symbolleiste wiedererkennt und so nutzen kann.

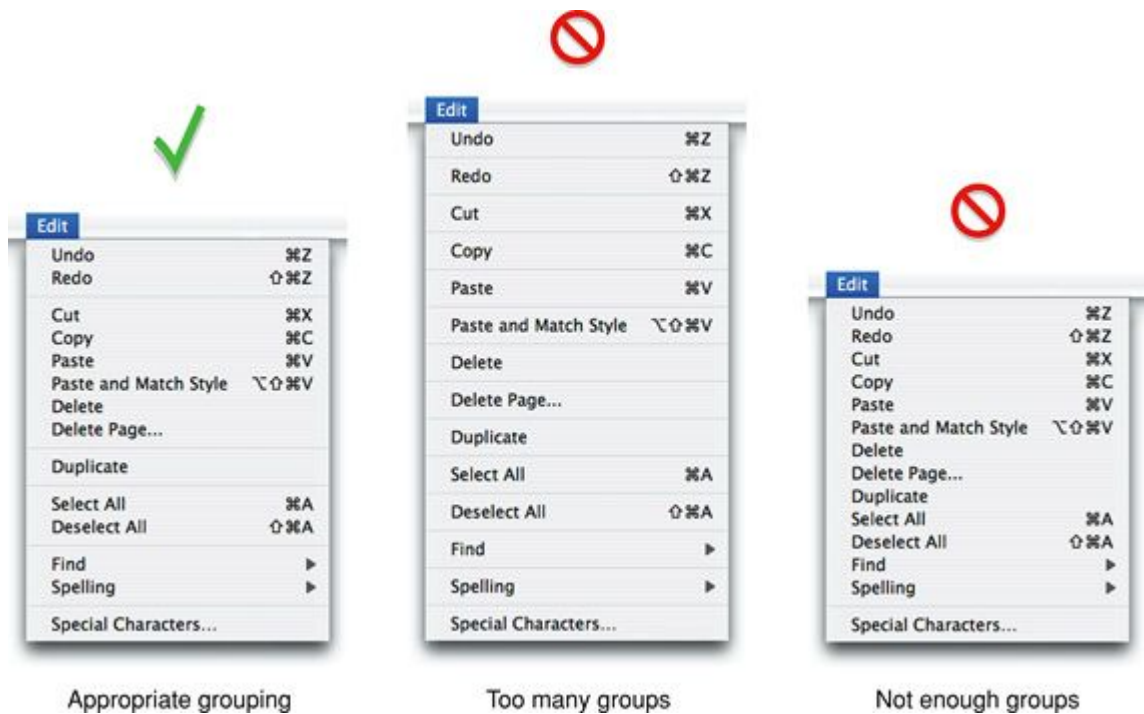


Abb. 11: Gruppierung von Menüeinträgen, Quelle: [Appl-01]

Bild 11 zeigt, wie thematisch gruppierte Menüeinträge die Übersichtlichkeit fördern. Eine weitere Möglichkeit ein Menü übersichtlich zu gestalten sind hierarchische Menüs wie in Abbildung 10.

Diese für textförmige Menüs geltenden Gestaltungsregeln sind im Rahmen dieser Arbeit sinngemäß auf textfreie grafische Menüs und insbesondere Anfasser zu übertragen.

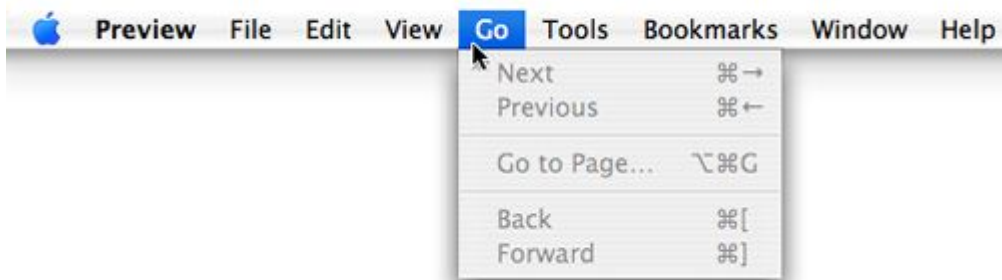


Abb. 12: Hauptmenü Reihenfolge, Quelle: [Appl-01]

Die Reihenfolge der Menüs wird ebenfalls im Styleguide festgelegt. So hat jede Anwendung über folgende Menüs (von links nach rechts) zu verfügen: Datei – Bearbeiten – Fenster – Hilfe. Weitere anwendungsspezifische Menüs sind zwischen „Bearbeiten“ und

„Fenster“ in logischer Reihenfolge anzuordnen. Ein entsprechendes Beispiel aus dem Styleguide zeigt Abb. 12.

Kontextmenüs sind definiert als Menübefehle, die – bei Windows über den rechten Mausklick – im Zusammenhang mit dem gewählten Objekt häufig genutzt werden. Die Menüinhalte sind die Übersichtlichkeit erheblich steigernd, vom jeweiligen Kontext dynamisch abhängig. So wird das Kontextmenü für einen markierten Text ein anderes sein als bei einem Desktop-Symbol.

Kontextmenüs haben sich in den letzten Jahren zu einem wesentlichen Element moderner Benutzungsoberflächen entwickelt. Da sie sich ansonsten wie normale Menüs verhalten sollen gibt der Styleguide keine weiteren Anweisungen zu ihrer Verwendung. Dieser Gedanke lässt sich sehr gut auf Anfasser für direkte Manipulation übertragen, die im Ansatz dieser Arbeit die Nutzungseffizienz entscheidend verbessern sollen.

Ausführlich behandelt der Styleguide die Gestaltung von Fenstern. Obwohl sie für den Benutzer möglichst gleich aussehen sollen, unterscheiden sie sich für den Entwickler in zahlreiche Fenstertypen, so z.B. in solche mit oder ohne Titelleiste, maximierfähige Fenster oder Dialogfenster.

Wesentlich für diese Arbeit erscheint vor allem die Definition der Fenstertypen:

- Das Anwendungsfenster ist das übergeordnete Fenster einer Anwendung. Es stellt Hauptmenüs, Symbol- und Statuszeilen zur Verfügung. Weitere Fenster mit den Benutzerdaten erscheinen in diesem Anwendungsfenster.
- Dokumentenfenster zeigen die Daten, die vom Benutzer selbst erstellt und in einer Datei gespeichert werden. Dokumentenfenster visualisieren also die Daten in einer verständlichen Form.
- Werkzeugfenster stellen Werkzeuge für den Benutzer zur Verfügung. Sie können über der Anwendung „schweben“ oder an die Anwendung „angedockt“ werden. Der Inhalt dieser Werkzeugfenster kann kontextabhängig gestaltet werden.
- Dialogfenster dienen dem Dialog mit dem Benutzer. Warnungen, Fehler und sonstige Meldungen werden dem Benutzer in diesen Fenstern angezeigt und erwarten eine Bestätigung. Auch Formulare sind Dialogfenster.

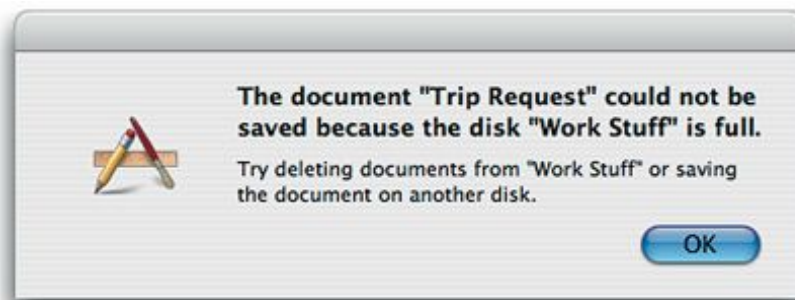


Abb. 13: Beispiel eines Dialogfensters, Quelle: [Appl-01]

Interessant für die Ziele dieser Arbeit sind die Angaben zu korrekten Dialogmeldungen, speziell zu Fehlermeldungen bei denen sich Texte möglicherweise nicht immer vermeiden lassen. Anstelle nichts sagender Fehlermeldungen („Fehler beim Schreiben der Datei“) sollen zielführende Fehlermeldungen eine Lösung des Problems ermöglichen. Ein Beispiel aus [Appl-01], wie sie auch bei Windows-Programmen wünschenswert wäre, zeigt Abb. 13.

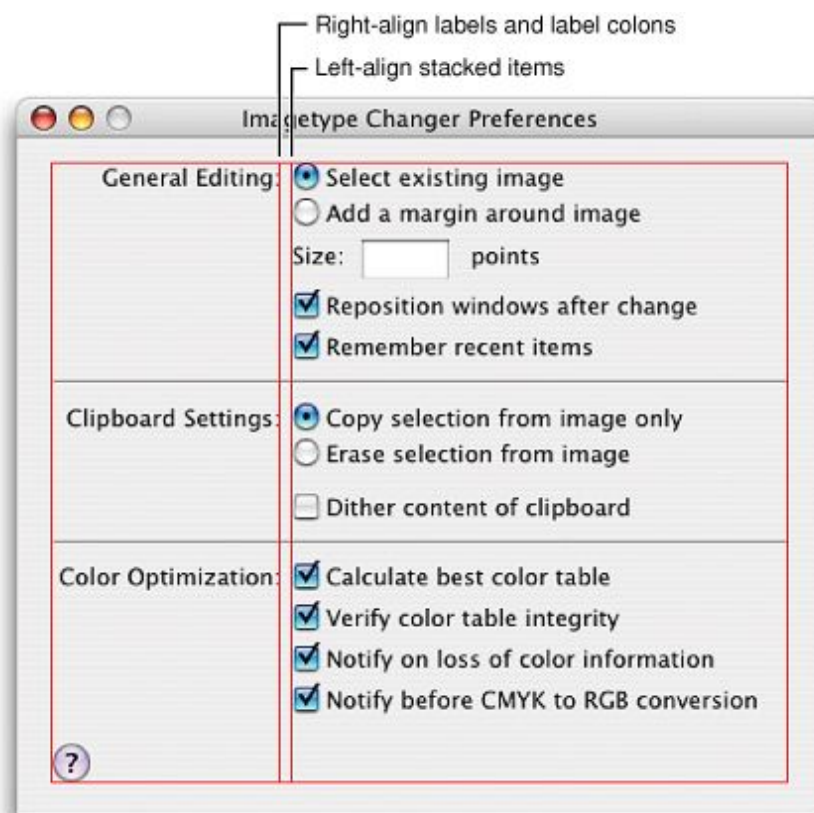


Abb. 14: Styleguide-Anwendungsbeispiel, Quelle: [Appl-01]

Abschließend gibt der Styleguide lehrreiche Beispiele mit vollständigeren Anwendungen, die die einheitliche Verwendung von Symbolen, Designs, Texten sowie der Ausrichtung vertiefen (siehe Abb. 14).

Apples Styleguide ist de-facto ein umfassendes Kompendium für den Gestalter von grafischen Benutzungsoberflächen. Die ersten Kapitel sind aufgrund ihres allgemein beschreibenden Charakters firmenunabhängiges Handwerkszeug für jeden Programmentwickler auch von Bausoftware. Weitere Kapitel dienen Apple-spezifischen Regeln. Dennoch beinhalten auch sie Informationen, die für die Entwickler von anderen Betriebssystemen zur Adaption und zur einheitlichen Gestaltung anregen.

So schafft es Apple durch die präzisen Vorgaben in seinem Styleguide ein weitgehend einheitliches Erscheinungsbild der Software auch von Dritten zu erzielen – mit der Folge, das auch neue Software aufgrund der bekannten Elemente direkt und intuitiv genutzt werden kann.

2.2.3 „Windows User Experience“

Der offizielle „Guideline for User Interface Developers and Designers“ [MS-01] von Microsoft ist im Internet veröffentlicht, allerdings nur als Kopie eines veralteten Buchs, welches nicht mehr publiziert wird. Die Angaben basieren auf Windows Version 98 bzw. 2000. Die neue Version Windows XP – und somit Microsofts „Aqua“-Oberfläche – fehlt. Anfang 2007 ist zudem der XP-Nachfolger „Vista“ mitsamt neuer Benutzungsoberfläche „Aero“ erschienen, womit die Guidelines unbrauchbar und obsolet werden.

Für die Zukunft wesentliche Unterschiede zu den Richtlinien von Apple bestehen nicht, zumal erklärende Bilder oder API-Befehle, mit denen die Operationen in Windows ausgeführt werden können, fehlen.

Zu Menüs und Symbolleisten gibt Microsoft ähnliche Hinweise wie Apple.

Einzig die Ausführungen zu Kontextmenüs sind in Bezug auf die sinnvolle Anordnung der Menüs ausführlicher als bei Apple. So sollen Kommandos wie Öffnen, Abspielen und

Drucken zuerst genannt werden, danach Befehle zum Kopieren und Verschieben, gefolgt von anderen Befehlen. Ebenso findet sich eine ausführliche Auflistung gängiger Menübefehle für verschiedene Menüs, die von Entwicklern genutzt werden können.

Für Menüs wird darüber hinaus eine entsprechende Wortwahl vorgeschlagen. So sollen, wenn das Menü als Verb formuliert ist, die Menüeinträge als Substantiv formuliert werden (im Menü „Einfügen“ die Einträge „Text“, „Tabelle“, „Bild“) bzw. entsprechend umgekehrt (im Menü „Tabelle“ die Einträge „Tabelle einfügen“, „Spalte auswählen“, „Zeile einfügen“).

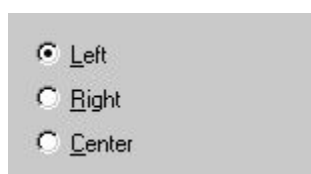


Abb. 15: Options-Schaltfläche
Quelle: [MS-01]

Neben Menüs werden Schaltflächen und deren Verwendung erläutert. Mit sog. Options-Schaltflächen (Option-Buttons oder Radio-Buttons) werden sich gegenseitig ausschließende Elemente ausgewählt. So kann ein Text mit dem Dialogfenster in Abb. 15 nur linksbündig, rechtsbündig oder zentrisch ausgerichtet werden.

In entsprechender Weise werden im Styleguide die gängigen Verhaltensweisen für die verschiedenen Schaltflächen vorgestellt, damit der Benutzer in jeder Anwendung dasselbe Verhalten von Schaltflächen voraussetzen darf.

Diese Merkmale sind wertvolle Hinweise, wie eine international verständliche, da überall gewohnte, Benutzungsoberfläche zu gestalten ist. So ist der Benutzer daran gewohnt, Menübefehle in der beschriebenen Reihenfolge und dem erläuterten Verhalten vorzufinden. Eine international verständliche, textfreie Benutzungsoberfläche muss denselben Prinzipien gehorchen.

Im vorletzten Kapitel widmet sich Microsoft ausführlich der visuellen Gestaltung und schickt dabei grundsätzliche Gestaltungsrichtlinien und Hinweise voraus. Diese sind derart elementar, dass sie zu Beginn des Styleguides hätten vorgestellt werden müssen.

Im Folgenden werden nur noch diejenigen Richtlinien von Microsoft erörtert, die die Richtlinien von Apple sinnvoll ergänzen oder im Widerspruch dazu stehen. Die Marktmacht von Microsoft kann im Einzelfall auch für diese Arbeit eine Übernahme der Lösung ratsam machen.

Der Microsoft Styleguide behandelt so ein wichtiges Thema, das von Apple nicht beachtet wurde: Die sog. „Lokalisation“, also die Übersetzung und landestypische Anpassung einer Anwendung, einem Zielproblem dieser Arbeit.

Die Lokalisation muss von Anfang der Softwareentwicklung an beachtet und berücksichtigt werden. So haben andere Länder und Kulturen nicht nur andere Sprachen und ggf. eine andere Schreibrichtung, sondern auch ein anderes Verständnis für bildliche Metaphern oder Soundeffekte.

Häufigstes Problem ist dabei, dass bei der Übersetzung einzelne Wörter und Sätze länger werden können als in englischer Sprache. Speziell bei der Beschriftung von Eingabefeldern sind die Elemente so anzuordnen, dass auch ein längerer Text Platz findet (siehe Abb. 16).

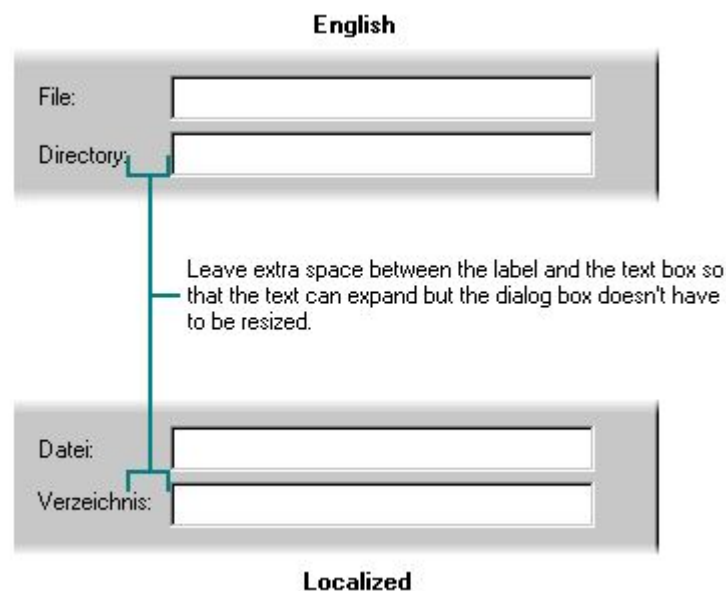


Abb. 16: Designkriterien für unterschiedliche Wortlänge bei der Übersetzung in Fremdsprachen
Quelle: [MS-01]

Ebenso können Wörter mehrere Bedeutungen haben, so z.B. „Zeit“, was als Tageszeit oder Dauer verstanden werden kann.

Die durchgängige Verwendung von Symbolen anstelle von Text würde dieses Problem lösen. Doch auch hierbei gilt es eine Form zu finden, die interkulturell verstanden wird und Missverständnisse oder Obszönitäten vermeidet.

So ist das in Amerika verbreitete Symbol eines Briefkastens mit Fähnchen für ein Mailprogramm verständlich, Europa hingegen präferiert die Darstellung eines Briefumschlages. Das in Amerika bei Fußgängerampeln als Stoppsignal verbreitete Symbol einer ausgestreckten Hand ist in asiatischen Ländern eine beleidigende Geste, also nicht weltweit verwendbar.

Darüber hinaus gilt es die Unterschiede bei Tastaturen (Vermeidung von nicht verfügbaren Zeichen in Tastaturkürzeln), den verschiedenen Zeichensätzen sowie den unterschiedlichen Formaten für Datum, Zeit, Währung oder Listentrennzeichen zu berücksichtigen, ebenso natürlich die bei arabischen oder persischen Sprachen umgekehrte Leserichtung.

2.3 Analyse marktbeherrschender Software und Bausoftware

2.3.1 Erkenntnisse aus Betriebssystemen

Windows XP ist derzeit der de-facto Standard für Benutzungsoberflächen. Da Windows-PCs in der Baupraxis gegenüber Apple- und Unix/Linux-Systemen überwiegen, berücksichtigt diese Arbeit das Windows-Betriebssystem besonders stark.

Die eingehende Analyse des Betriebssystems Windows XP ergab gegenüber den in Kapitel 2 erörterten Guidelines grundsätzlich keine neuen Erkenntnisse, die für die Ziele dieser Arbeit wesentlich wären.

Windows XP gibt insgesamt viele wesentliche Merkmale mit, die eine auf diesem Betriebssystem ausgeführte Anwendung für weltweiten Einsatz ebenfalls aufweisen sollte. Sei es die Optik der Darstellung (Gestaltung der Icons, Menüleisten, Farben), die intuitive Handhabung durch Anleihen an der Realität (3D-Optik, Sammelfunktion) oder die Individualisierbarkeit durch den Benutzer.

2.3.2 Erkenntnisse aus Textverarbeitungsprogrammen

Microsoft Word, ein Textverarbeitungsprogramm, ist Teil von Microsofts „Office 2003“-Paket, dem Marktführer bei Büroanwendungen, auch im Bauwesen. Von der Text-

verarbeitung über Tabellenkalkulation hin zu Vortragspräsentationen finden die Programme des Office-Paketes auch in Firmen und Büros der Baubranche Anwendung. Das Beherrschen von Microsofts Office-Paket gehört selbst für Berufsanfänger bereits zum Standard, dementsprechend bekannt ist die Benutzung und Benutzerführung von Office – derart bekannt, dass sie aus Gründen der Lerneffizienz als Vorgabe für weitere Programme des Bauwesens angesehen werden muss, also auch für diese Arbeit.

Microsoft Word verfügt über mehrere typische Merkmale intelligenter Benutzungsoberflächen, so wie Sie Jef Raskin in [Rask00] beschreibt. So gilt es als Ziel, dass sich Software möglichst naturgetreu verhält, z.B. ein Cursor in Scheren-Darstellung, wenn die Operation „Bild beschneiden“ lautet. Ein weiteres Ziel ist es, dass sich Software den persönlichen Vorlieben des Benutzers anpassen lässt.

Ein typischer Fall der Textverarbeitung ist das Einfügen einer Tabelle. Drei Möglichkeiten, dieses Ziel zu erreichen, seien nachfolgend als Musterfälle im Sinne dieser Arbeit diskutiert.

1) Tabelle zeichnen

Über den textförmigen Menübefehl „Tabelle zeichnen“ erscheint ein Zeichenstift, mit dem interaktiv horizontale und vertikale Linie auf das Arbeitsblatt zu zeichnen sind. Ähnlich der natürlichen Verhaltensweise, mit einem Stift und einem Lineal eine Tabelle auf ein Blatt Papier zu zeichnen, kann so eine Tabelle in Word gezeichnet werden (siehe Abb. 17). Das händische Zeichnen einer Tabelle und die freie Gestaltung der Zellenaufteilung bedingt, dass Spaltenbreiten und Zeilenhöhen variieren und nicht regelmäßig sind (siehe Abb. 17). Eine maßlich exakte Anpassung ist durch späteres Formatieren der Zellen möglich.

Dabei ist das System so „intelligent“, die frei gezeichneten Linien miteinander zu verknüpfen. Mit vertikalen Linien werden so Spalten, mit horizontalen Linien Zeilen generiert.

Dieser Ansatz wäre ungemein geeignet für die Definition von Gebäuderastern in Bau-Software wie noch gezeigt wird.

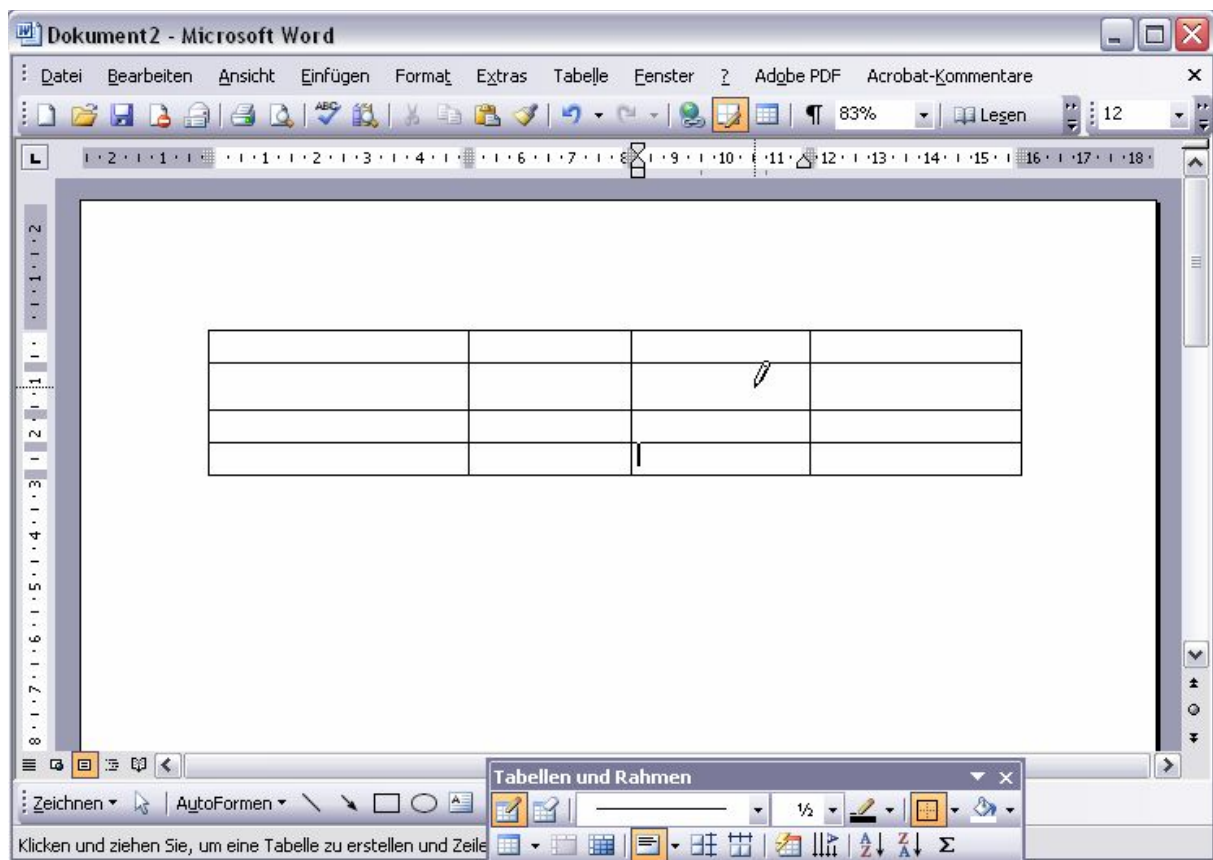


Abb. 17: "Tabelle zeichnen" - Freihändig gezeichnete Tabelle in Word

Der Mauszeiger nimmt dabei eine metaphorische Darstellung als Zeichenstift an. Die Linien werden während des Zeichnens, solange die Maus noch nicht losgelassen wurde, sichtbar mit der Maus mitgezogen und stellen sich automatisch senkrecht bzw. waagrecht ein, wie z.B. in Rasterlinien im Bauwesen (siehe Bild 18). Ebenso verhält es sich für den Entwurf von Schrauben- und Lochbildern bei der Detaillierung von Anschlüssen, die senkrecht und waagrecht zur Stabachse angeordnet werden.

Aufgrund dieser bildlichen Darstellung ist die Funktion „Tabelle zeichnen“ auch für unerfahrene Benutzer intuitiv und ohne Schulungsaufwand verständlich, wie auch sehr effiziente Lehrveranstaltungen im Fach Bauinformatik an der Bergischen Universität Wuppertal belegen.

Dieses Fallbeispiel ist im Sinne dieser Arbeit eine besonders vorbildliche Lösung, also tatsächlich ein Musterfall einer international unmittelbar verständlichen, sprach- und text-

freien Benutzungsoberfläche.

Für das Bauwesen ist diese Musterlösung von besonderem Interesse. Bei genauer Betrachtung fällt nämlich auf, dass das Eingabeproblem eines Gebäuderasters mit Rasterlinien und Knoten an Kreuzungspunkten von Rastern völlig analog zu Tabellen ist.

Reihen und Achsen entsprechen Zeilen und Spalten. Selbst die exakte maßliche „Formatierung“ könnte übernommen werden, ebenso die anschaulichen zeichnerischen Änderungsmöglichkeiten. Die Verbesserung der Effizienz und Verständlichkeit wäre mit diesem Ansatz erheblich, wie Azadnia in [Azad07] im Praxisversuch nachgewiesen hat.

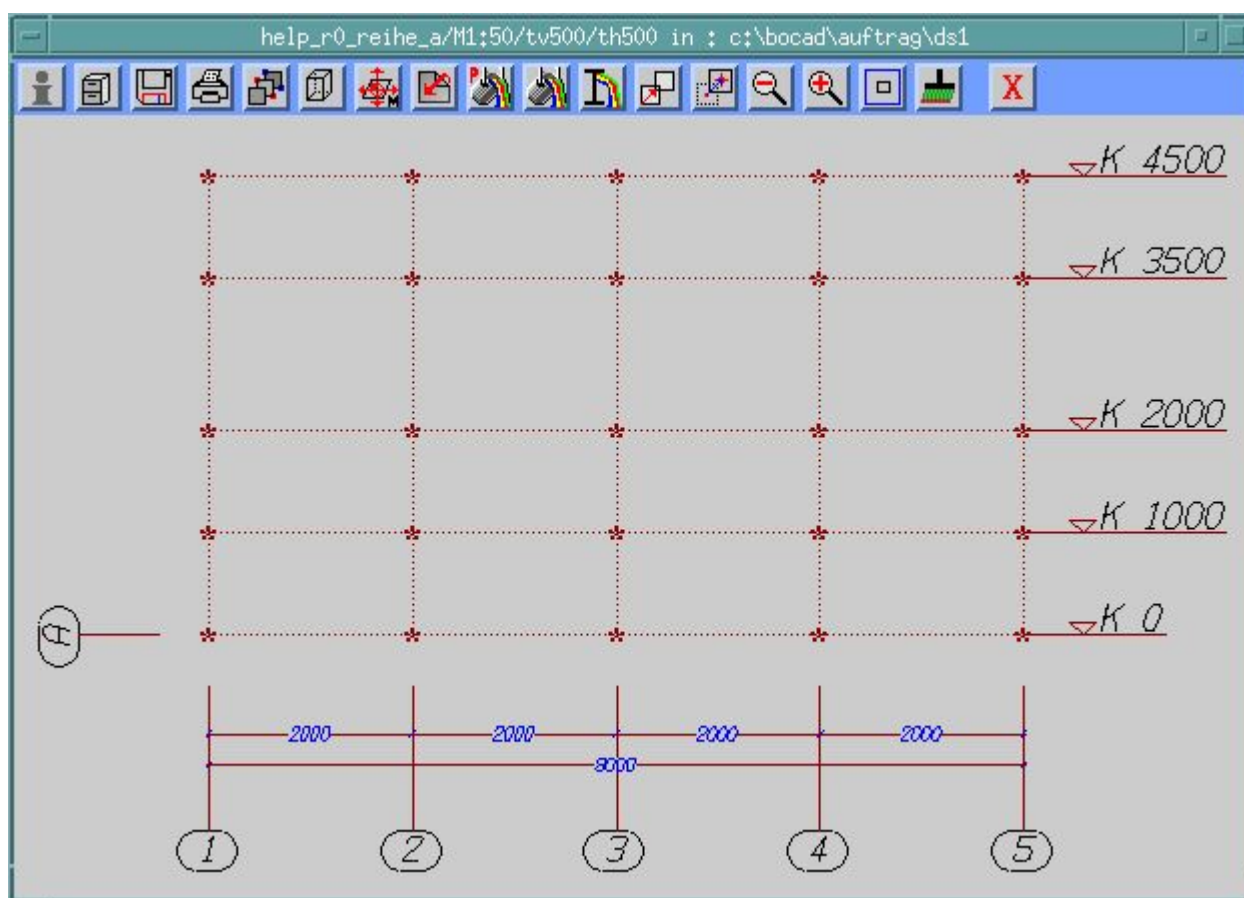


Abb. 18: Rasterlinien in BOCAD-3D Version 19

2) Einfügen einer Tabelle über ein Formular

Entsprechend [EN-9241] gehören auch Bildschirmformulare zur metaphorischen Darstellung, mit Entsprechung bei den bekannten Formularen in Schriftform. Dabei ist der Begriff „Formular“ in der Informatik nicht gebräuchlich. Es wird stattdessen der Begriff „Dialogfenster“ verwendet.

So wird über den Menüeintrag „Tabelle – Einfügen – Tabelle“ ein entsprechendes Dialogfenster geöffnet (siehe Abb. 19).

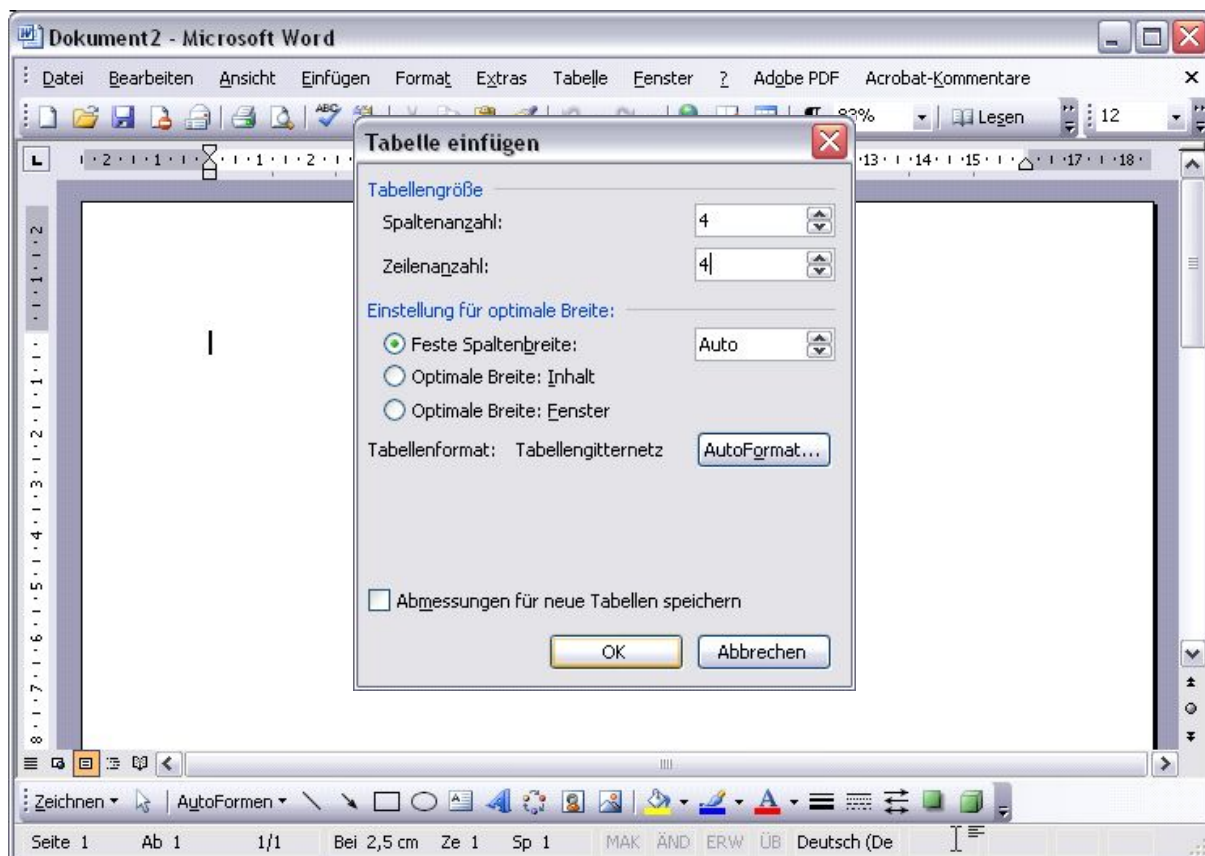


Abb. 19: Dialogfenster "Tabelle einfügen"

Hier gibt der Benutzer die gewünschte Spalten- und Zeilenanzahl sowie weitere Einstellungen ein. Mit Klick auf die „OK“-Schaltfläche schließt sich das Dialogfenster und die Tabelle wird automatisch erstellt.

Der Vergleich beider Vorgehensweisen zeigt, dass sie zwar zum gleichen Ergebnis führen – eine Tabelle wird eingefügt – sich aber in der Sprach- und Textfreiheit signifikant unterscheiden. Lösung 1 erreicht dieses Ziel mit Ausnahme des Menüeintrages vollständig sprachfrei und höchst anschaulich, Lösung 2 ist ohne landesspezifische Übersetzungen weltweit unverständlich.

Während bei „Tabelle zeichnen“ sogar ohne Zahlenangaben die Anzahl der Zeilen und Spalten der Tabelle interaktiv bestimmt wird, muss bei „Tabelle einfügen“ vorab bekannt sein, wie die Tabelle letztendlich aufgeteilt sein soll. Später ist eine Veränderung der Ta-

bellenaufteilung zwar über die Funktion „Zellen teilen“ möglich, was aber ebenso unanschaulich ist. Lösung 2 wird daher verworfen.

3) Einfügen mittels Symbol

Der vorbildlichen Lösung 1, die weltweit verständlich ohne Sprache und Text auskommt, fehlte zur Perfektion nur noch der Aufruf über ein Symbol statt über einen Menüeintrag. Diese Chance hat Microsoft vertan, da das zum Einfügen von Tabellen bestehende Symbol eine dritte Methode anbietet.

Dabei ist diese Möglichkeit vom Benutzer selbständig, ohne Schulungsaufwand oder Lesen einer Dokumentation erlernbar. Entsprechend den Forderungen von [EN-9241] zur Erreichung der Lernförderlichkeit werden für bestimmte Funktionen durchgängig dieselben Symbole verwendet.

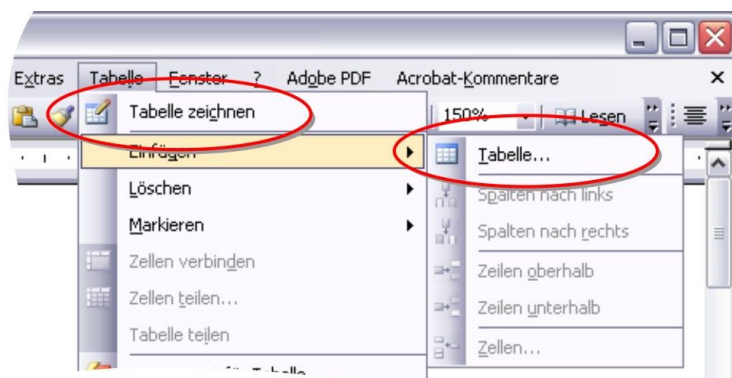


Abb. 20: Durchgängige Verwendung eines Symbol

So erkennt der Benutzer bei Verwendung der Menüeinträge, dass immer ein identisches, nur im Detail abgewandeltes Symbol für eine Tabelle verwendet wird (Abb. 20). Sehr leicht wird er so selbstständig auf ein identisches Symbol in den Symbolleisten aufmerksam. Beim Überfahren des Symbols mit der Maus zeigt ihm die Quickinfo an, dass diese Schaltfläche eine Tabelle einfügt (siehe Abb. 21).

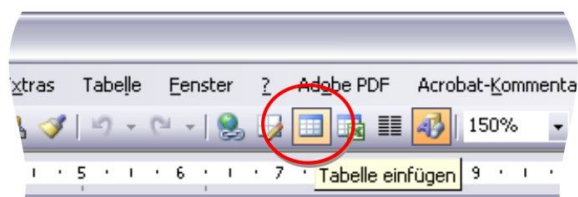


Abb. 21: Symbolleiste "Tabelle einfügen"

Mit Klick auf die Symbolschaltfläche ermöglicht ein sprachfreier Dialog in Form von 5 Zeilen und Spalten das Erstellen einer Tabelle (siehe Abb. 22). Intuitiv versteht der Benutzer, dass er über diesen einfachen Dialog die Zellenaufteilung bestimmen kann, die durch grau eingefärbte Zellen angezeigt wird. Mit nur zwei Mausklicks kann so eine Tabelle eingefügt werden.

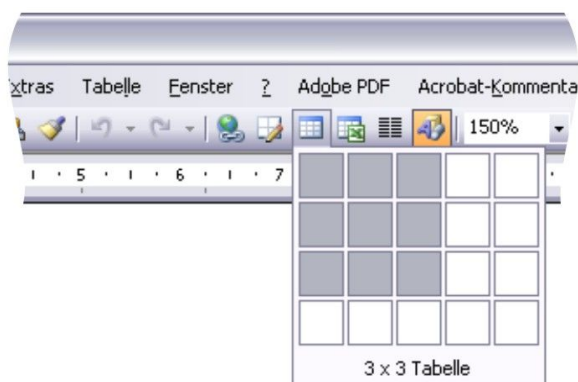


Abb. 22: Einfügen einer Tabelle mittels Symbolschaltfläche

Bei dieser Form des Benutzerdialogs muss weder ein verschachteltes Menü oder ein Dialogfenster geöffnet werden. Der Dialog erreicht so eine sehr hohe Arbeitsgeschwindigkeit. Zudem kann der Dialog vollkommen sprachfrei erfolgen, ohne dass der Benutzer Einschränkungen im Bedienungskomfort hinnehmen müsste.

Microsoft Word kann bezüglich dieser einen Funktion „Tabelle einfügen“ als Musterbeispiel für drei wesentliche Kriterien bei der Gestaltung von direkten, sprachfreien, intuitiv verständlichen Benutzungsoberflächen angesehen werden: Die Verwendung natürlicher Verhaltensweisen in Form von metaphorischen Darstellungen aus der Realität, verschie-

denen Wege ein Ziel zu erreichen und der sprachfreien und intuitiv erlernbaren Benutzerdialoge.

Mit dieser Auswahlmöglichkeit aus drei Wegen eine Tabelle einzufügen, kommt Word zwar der zweiten Forderung der [EN-9241] nach, Software seinen persönlichen Vorlieben entsprechend anpassen zu können. Dennoch muss speziell diese Forderung der Norm diskutiert werden.

So sind mehrfache Alternativen, eine Funktion aufzurufen, ein typisches Problem historisch gewachsener Softwaresysteme [Rask01]. Mit jeder neuen Version werden neue Funktionen implementiert, ohne dass die alten, überholten Varianten deaktiviert werden. Dies wird als benutzerfreundlich angesehen, da sich so die etablierten Verhaltensweisen auch in der neuen Softwareversion nutzen lassen. Dennoch wird so nur eine den Benutzer verwirrende Vielzahl an Wegen geschaffen, ein Ziel zu erreichen. So ist eine vollständig reflexhaft beherrschbare, erstklassige Lösung besser lernbar als mehrere, unterschiedliche Alternativen geringerer Effizienz.

Die Praxiserfahrungen mit effizientesten Spitzenanwendern des CAD-Hochleistungssystems BOCAD-3D bestätigen dieses Problem geradezu frappant. Blieben sie reflexhaft beherrschten alten Wegen verhaftet, obwohl in neuen Versionen effizientere Wege angeboten wurden, sank ihre Leistung unter den Durchschnitt aller Anwender.

2.3.3 Erkenntnisse aus Zeichenprogrammen

Als Beispiel für ein professionelles Grafikbearbeitungsprogramm wird CorelDraw 12 aus dem kanadischen Softwarehaus Corel gewählt. Basierend auf Vektorgrafik ermöglicht es dem Benutzer, komplexe Zeichnungen zu erstellen. Es ist von der Arbeitsweise mit 2D-CAD-Programmen wie Autocad zu vergleichen. So wird CorelDraw vielfach auch in Ingenieurbüros eingesetzt, vor allem im Bereich der Stadt- und der Infrastrukturplanung. Hier ersetzt es klassische CAD-Programme zur Gestaltung von Entwürfen. Aufgrund dessen und dem im Folgenden vorgestellten, innovativen Ansatz der Benutzungsoberfläche zeichnet sich CorelDraw besonders für diese Arbeit aus.

CorelDraw gilt als ein Musterfall für die von Chang in [Chan02] geforderten und entwickelten „intelligenten Objekte“. Im folgenden seien zwei als „intelligent“ zu bezeichnende Leistungen aus CorelDraw ausgewählt und diskutiert.

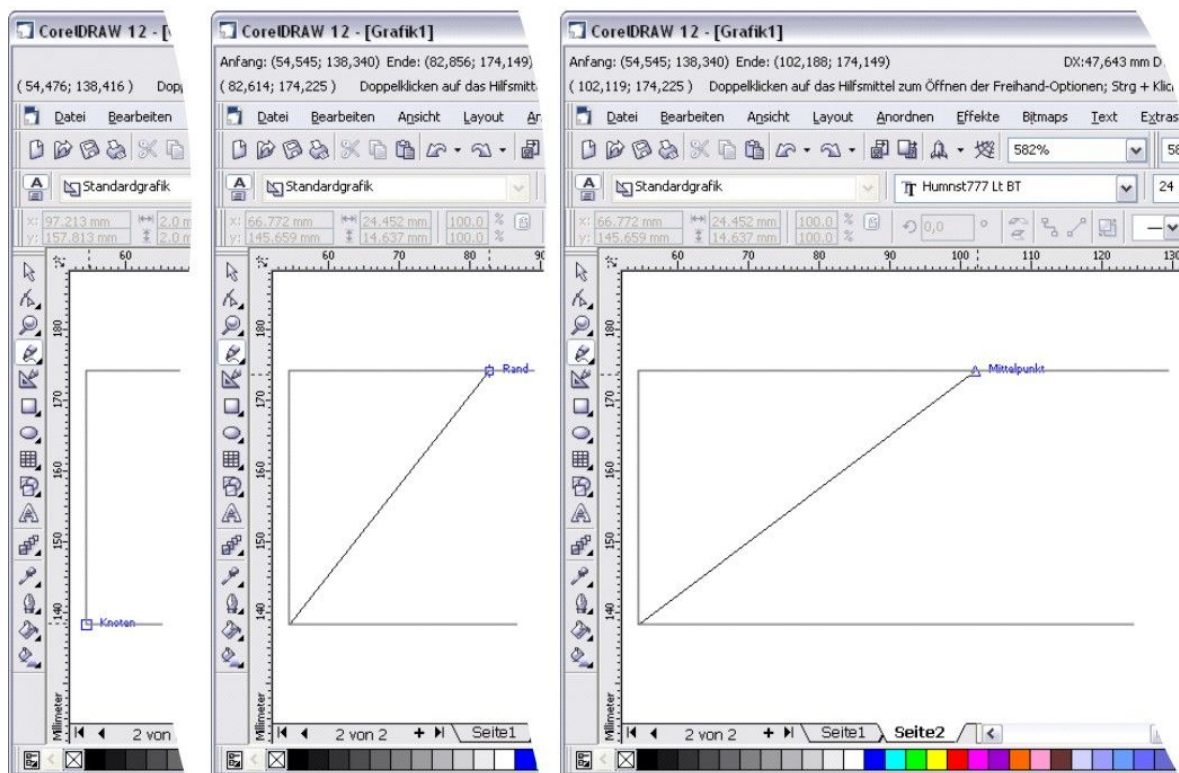


Abb. 23: „Intelligenter“ Objektfang in CorelDraw 12

„Intelligente Objekte“

In CorelDraw werden die Objekte nicht als Pixelfolge, sondern in Form von Vektoren beschrieben. Für eine Linie sind die X- und Y-Koordinaten des Vektoranfangs- und Endpunktes zu bestimmen. Für einen Vollkreis muss der Kreismittelpunkt sowie der Radius angegeben werden. Diese Objekte können dann mit Objekteigenschaften bzgl. Linienart oder Füllung versehen werden.

Als Beispiel für die „Intelligenz“ von Objekten, die für Sprach- und Textfreiheit von Benutzungsoberflächen notwendig ist, sei hier das Zeichnen einer weiteren Linie innerhalb eines bereits auf der Zeichenoberfläche befindlichen Rechtecks demonstriert. Die Linie soll von der unteren linken Ecke des Rechtecks bis zum Mittelpunkt der Oberkante geführt werden.

Durch den integrierten „Objektfang“ entsprechend Abb. 23 „fängt“ die Maus beim Nähern an die untere linke Ecke des Rechtecks den Knoten und zeigt dies durch eine Veränderung des Mauszeigers und eine Onlinehilfe an. Nachdem so der Startpunkt festgelegt wurde, kann der Zielpunkt der Linie mit der gleichen „Intelligenz“ mittels Objektfang ausgewählt werden. Sobald der Mauszeiger in die Nähe eines sog. „Hotspots“ kommt, zeigt der Objektfang dies durch eine Cursorveränderung an (siehe mittlerer und rechter Ausschnitt in Abb. 23).

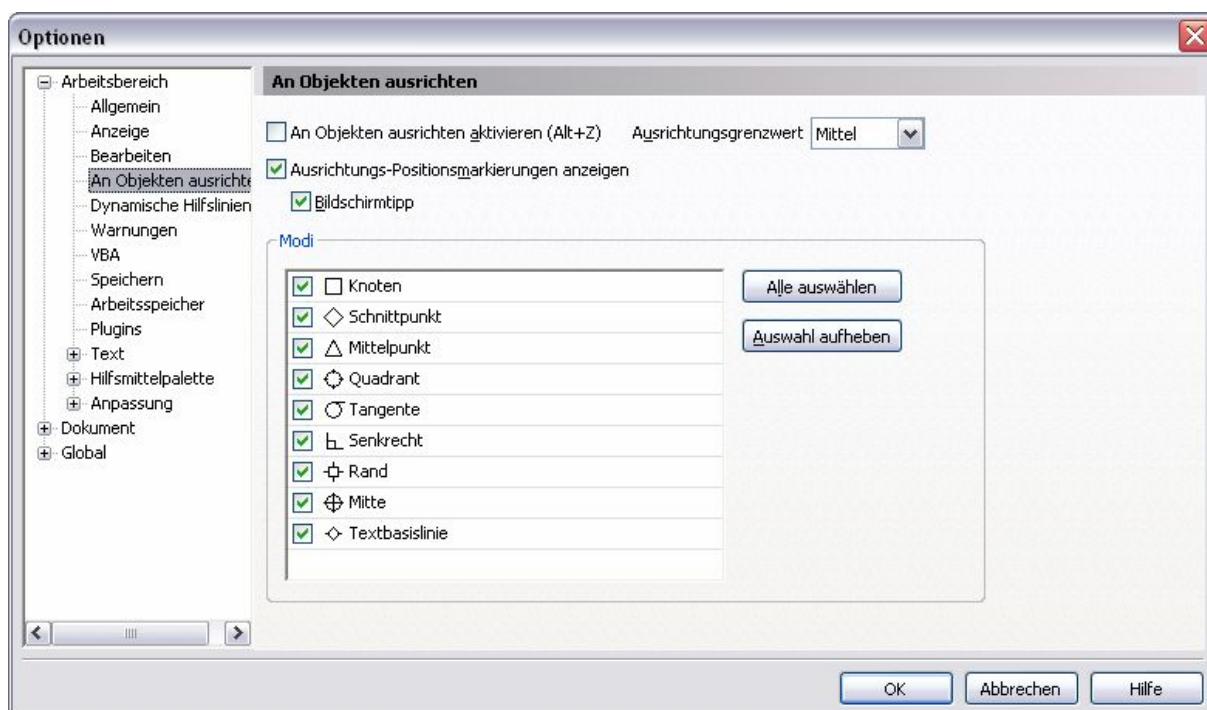


Abb. 24: Individualisierbarkeit des Objektfangs bei CorelDraw 12

Während das diskutierte Freihandzeichnen von Tabellen in Word zu nachträglichen Maßangaben zwingt, werden hier funktional sinnvolle Maße „intelligent“ angeboten zur direkten Manipulation.

Für den Benutzer bedeutet dieser Objektfang eine wesentliche Arbeitserleichterung, da kein falsches Objekt bzw. bei nah beieinander liegenden Objektkanten nicht die falsche Kante ausgewählt wird.

Auch CorelDraw 12 entspricht den Forderungen der [EN-9241] nach Individualisierbarkeit durch den Benutzer, da das Verhalten der Hotspots des Objektfangs entsprechend individualisiert werden kann (siehe Abb.24).

„Intelligente Symbolleisten“

Symbolleisten können auf Wunsch auf der Benutzungsoberfläche platziert werden (siehe Abb. 25). Die Symbolleiste „Objekteigenschaften“ passt sich intelligent dem aktiven Objekt an und ist bis auf Maßzahlen sprach- und textfrei.

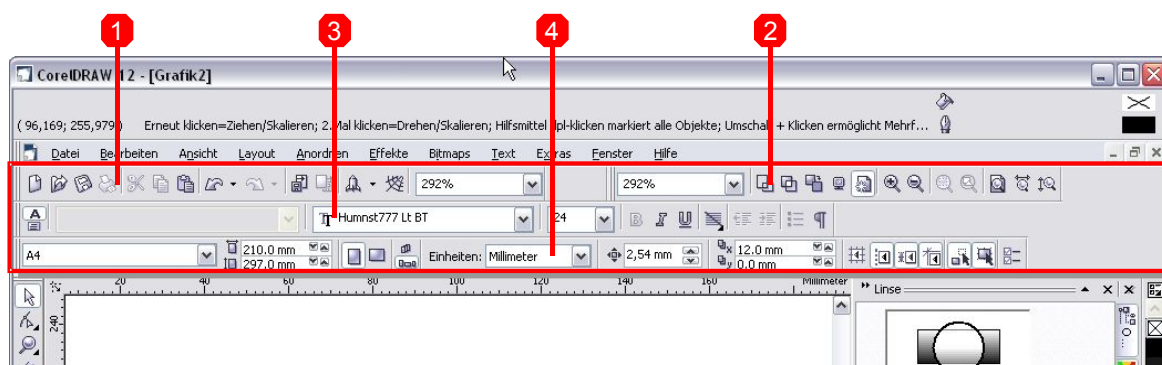


Abb. 25: CorelDraw 12 Symbolleisten (1=Datei, 2=Zoom und Ansicht, 3=Texteigenschaften, 4=Objekteigenschaften)

Abb. 26 zeigt vier Zustände der Symbolleiste „Objekteigenschaften“ - je nachdem, welches Objekt gerade aktiviert wurde. Eine einzige Symbolleiste informiert so über die Eigenschaften aller im aktiven Dokument verwendeten Objekte. Die bei ähnlichen Objekten identischen Eigenschaften wie Position, Größe, Skalierungsfaktor u.ä. sind immer an derselben Stelle angeordnet.

Vor allem die durchgängige und durchdachte Anordnung der Elemente innerhalb der Symbolleiste trägt maßgeblich zur intuitiven Verständlichkeit bei.

Jede Symbolleiste beginnt mit den in allen Objekten vorhandenen Eigenschaften. Danach folgen die Eigenschaften, die nur bei speziellen Objekten anzuzeigen sind.

Der Benutzer wird daher schnell und ohne Suche die Eingabefenster zum Skalieren eines Objektes finden – egal welches Objekt gerade aktiviert ist.

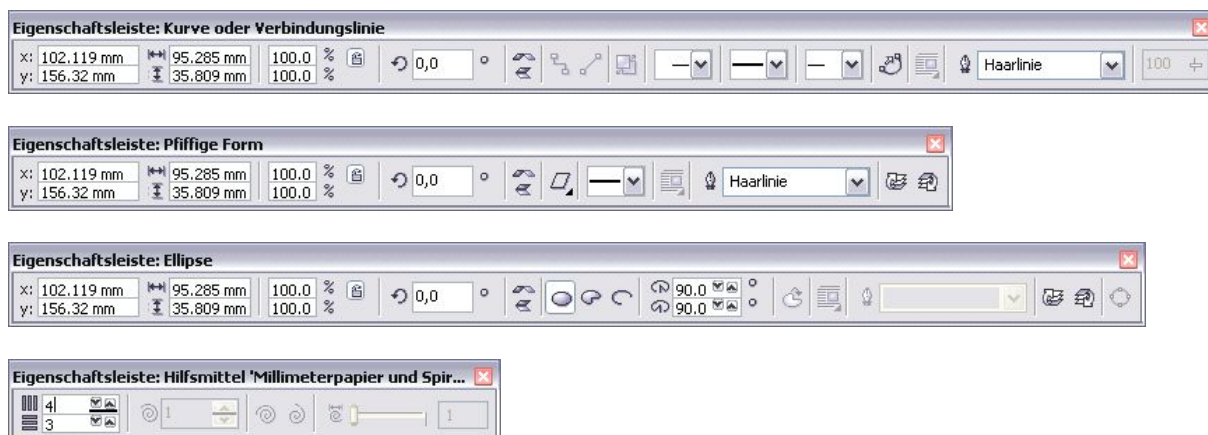



Abb. 26: Symbolleiste mit "Objekteigenschaften" verschiedener Objekte

Der gleiche Arbeitsprozess ist bei Microsoft Word grundlegend anders gelöst. Hier gibt es kein zentrales Element in der Benutzungsoberfläche, welches einheitlich die Eigenschaften von z.B. Tabellen, Grafiken oder Textabschnitten regelt. Für jede Änderung der Eigenschaften ist ein eigener Arbeitsschritt notwendig, noch dazu mit jeweils unterschiedlichen Dialogfenstern. Die Übernahme eines derart unsystematischen, weder effizienten noch verständlichen Ansatzes wird daher für diese Arbeit verworfen.

Wie auch in Word eine ähnliche Symbolleiste für die Tabelleneigenschaften zu verwenden wäre, zeigt Bild 26 unten. In CorelDraw ändert diese Symbolleiste die Anzahl der Spalten und Zeilen für die sog. „Millimeterpapier“-Funktion.

Es fällt zudem eine hier wesentliche Eigenschaft der Symbolleisten auf. Trotz umfangreicher Änderungs- und Einstellungsmöglichkeiten ist die Symbolleiste bis auf die notwendigen Zahlen textfrei.

Die oberste Darstellung „Kurve oder Verbindungslinie“ aus Abb. 26 zeigt die kompakte Anordnung der einzelnen Schaltflächen und Eingabefelder. So kann z.B. die Position des Objektes mithilfe der X- und Y-Koordinaten oder die Größe in vertikaler und horizontaler Richtung sowohl in mm als auch in Prozent verändert werden. Das Vorhängeschloss  ermöglicht dabei ein automatisches proportionales Skalieren. Über unmittelbar verständliche Symbole ist eine Drehung in Grad möglich oder das horizontale und vertikale Spiegeln.

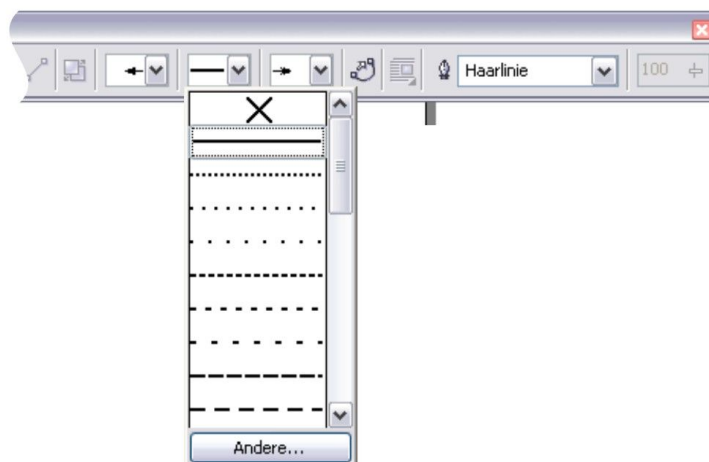


Abb. 27: Ausschnitt Symbolleiste „Eigenschaften“

Bild 27 erläutert, wie weitgehend textfrei und kompakt angeordnet für Objekte vom Typ „Kurve oder Verbindungslinie“, die per Definition offene Enden hat, die Linienart, die Strichstärke und für den Anfangs- und Endpunkt der Linie Pfeilspitzen ausgewählt werden. Das einzige Wort „Haarlinie“ steht für die dünnste darzustellende Liniendicke, ansonsten werden hier Maßangaben mit Dimensionen verwendet (z.B. 0,75 mm).

Es fällt auf, dass bei der Gestaltung der Dialogfelder pro Objekt nur die besonders häufig genutzten Eigenschaften aufgenommen wurden, um die Symbolleisten nicht unübersichtlich werden zu lassen. So fehlt in der Eigenschaftsleiste „Ellipse“ die Möglichkeit, Linienart und Strichstärke modifizieren zu können. Hier wurde zugunsten anderer, öfter zu modifizierender Eigenschaften darauf verzichtet. Eine Änderung der nicht in die Symbolleiste aufgenommenen Eigenschaften ist dann über das textförmige Menü möglich.

Diese intelligenten Objekte und ihre Symbolleisten sind eine vorbildliche Lösung im Sinne dieser Arbeit. Speziell für das Bauwesen ist die hier diskutierte Lösung von besonderem Interesse.

So ist das Erzeugen von Objekten und der automatische Objektfang vollkommen analog zum Konstruieren von Stützen, Riegeln und sonstigen Bauteilen in CAD-Systemen der Hochleistungs-klasse. Wird ein Riegel an eine Stütze angeschlossen, steigern Hotspots an den Eck- und Mittelpunkten der Stütze die Effizienz und Irrtumsfreiheit beim Platzieren des Riegels wesentlich.

Dem Benutzer werden nach Auswahl eines Elementes nur die für dieses Objekt relevanten Eigenschaften, meist textfrei in Form von intuitiv verständlichen Symbolen, angezeigt. Dies kann, wie bei CorelDraw vorgestellt, mit Symbolleisten geschehen, oder bei komplexeren Dialogen in Form von Dialogfenstern. Der programmtechnische Ansatz hierzu wird im weiteren Verlauf dieser Arbeit vorgestellt.

2.3.4 Erkenntnisse aus Website-Entwicklungswerkzeugen

Die relativ junge Entwicklungsumgebung Dreamweaver für Websites des amerikanischen Softwarehauses Adobe (vormals Macromedia) soll kurz vorgestellt werden, weil sie gemeinsam mit CorelDraw den weltweiten Industriestandard mitprägt.

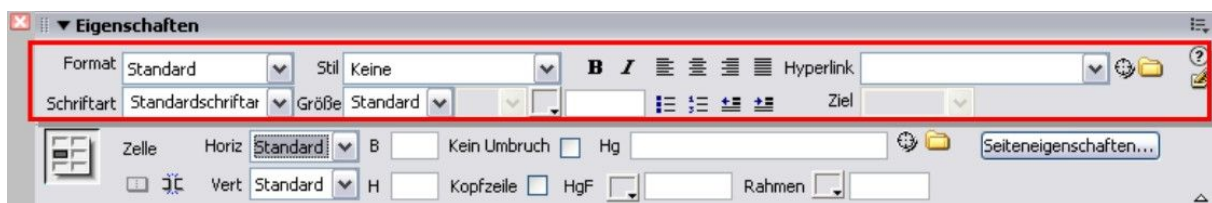


Abb. 28: Eigenschaften einer Tabellenzelle in Dreamweaver MX 2004

Abb. 28 zeigt die Symbolleiste der Eigenschaften einer Tabellenzelle. In der HTML-Programmierung wird zur Anordnung der Elemente einer Website mit sog. „blinden Tabellen“ gearbeitet, also Tabellen, die keine Gitternetzlinien haben und nur als Raster zur Anordnung dienen.

Die Analyse der in der Symbolleiste angebotenen Eingabe- und Auswahlfelder ergibt, dass ein Großteil der dort genannten Eigenschaften das Bedienfeld unnötig füllen, da sie für das Objekt irrelevant sind. So sind die in Bild 28 rot markierten Felder zum Formatieren von Text in dieser Gruppe nicht vonnöten. Ein Weglassen dieser Felder würde das gesamte Bedienfeld übersichtlicher machen. In diesem Punkt ist die Lösung von Adobe also nicht vorbildlich.



Abb. 29: Eigenschaften einer Grafik mit Hyperlink

Die beiden Abbildungen 29 und 30, in denen die Eingabefelder zur Eingabe eines Hyperlinks rot markiert wurden, verdeutlichen die unnötig unterschiedlichen Lösungen für Grafik und Text.

Das Einfügen eines Hyperlinks, also das Verknüpfen einer Webseite mit einer anderen, ist elementares Element bei der Gestaltung von Internetseiten und dementsprechend ein häufiger Arbeitsprozess. So ist es nicht zweckmäßig, dass sich das Eingabefeld in der Bedienfeldgruppe „Grafik“ (Abb. 29) an einer anderen Stelle als in der Bedienfeldgruppe „Text“ (Abb. 30) befindet.



Abb. 30: Eigenschaften eines Textes mit Hyperlink

Bei häufigem Wechsel zwischen dem Einfügen von Hyperlinks auf Grafiken und auf Text bedeutet dies einen ständigen Wechsel des Aufmerksamkeitsfokus des Benutzers auf einen anderen Teil des Bedienfeldes.

In der Bedienfeldgruppe „Text“ findet sich zudem auch ein Feld zur Eingabe der Zellenbreite und -höhe, sofern sich der Text in einer Tabelle befindet (in Abb. 30 grün markiert). Auch dieses Eingabefeld sollte an dieser Stelle nicht angezeigt werden, da es nicht zu den Texteigenschaften zählt.

Zudem kommt keines der Bedienfelder ähnlich textfrei aus wie bei CorelDraw. Selbst einfache Eingaben, wie für Breite und Höhe, sind sprachspezifisch mit „B“ und „H“ abgekürzt.

Die Lösungen von Adobe sind somit als wenig effizient und weltweit nur nach Übersetzung verständlich insgesamt zu verwerfen.

2.3.5 Erkenntnisse aus BOCAD-3D Stahl und Metallbau Version 19

Bei BOCAD-3D handelt es sich um ein CAD-Hochleistungssystem, das als Konstruktionsprogramm mit einem dreidimensionalen Produktmodell arbeitet. Im Gegensatz zu

weit verbreiteten Softwareprodukten wie Autocad, welches eher eine Zeichenhilfe darstellt, werden bei BOCAD-3D die Komponenten des Bauwerks als dreidimensionale Körper über intelligente CAD-Methoden gebildet. Mit diesen CAD-Konstruktionsmethoden werden Rahmenecken, Stützenfüße etc. automatisch vom CAD-System entsprechend den Benutzerwünschen konstruiert. So entsteht ein vollständiges, digitales Abbild des Bauwerks im Rechner, ein so genanntes Produktmodell.

Dieses Produktmodell des Bauwerks wird dann, typisch für CAD-Hochleistungssysteme, automatisch ausgewertet und weiterverarbeitet, d.h. Gleichteile werden erkannt, Positionsnummern nach Regeln vergeben, alle Zeichnungsarten nach Regeln automatisch erstellt, alle Stücklisten erzeugt und CNC-Steuerungsdaten für die automatische Fertigung generiert und weitergeleitet.

BOCAD-3D in Version 19 aus dem Jahr 2005 kann auf dem Betriebssystem Windows eingesetzt werden ebenso wie auf Linux-Systemen. Durch eine Emulation mit der Software Exceed der Firma Hummingbird wird BOCAD-3D auf Windows lauffähig. Diese Emulation ahmt nur unvollkommen ältere Windows-Versionen nach, ist also für Windowsbenutzer ungewohnt. Dadurch kann BOCAD-3D ohne Anpassung am Quellcode auch auf Linux-Systemen eingesetzt werden, ist also wie grundsätzlich anzustreben nicht plattformabhängig.

Auch wenn mittlerweile eine etwas zeitgemäßere Version 20 ausgeliefert wurde, benutzen noch viele Anwender die ausgetestete, stabile Version 19. Daher wird hier Version 19 analysiert und später mit Version 20 verglichen.

Im Folgenden seien vor Allem die Aspekte analysiert, die wesentlich von den bisher vorgestellten Industriestandards abweichen und ggf. einer umfangreichen Verbesserung zu unterziehen sind, um Effizienz und Verständlichkeit zu steigern. Der programmtechnische Ansatz dieser Arbeit und die vorgestellten Software-Prototypen werden entsprechende Neuerungen und innovative Lösungen aufzeigen.

Bei Beobachtung von Studierenden an der Bergischen Universität Wuppertal ergibt sich, dass die Benutzungsoberfläche von BOCAD-3D Version 19 (Bild 31) intuitiv nicht verständlich ist. In den semesterbegleitenden Übungen bereiten die Auswahl von Fenstern, das Aktivieren von Schaltflächen und Kontrollelementen sowie das Ausfüllen von Dialog-

fenstern anhaltend Schwierigkeiten, da sie nicht nach den hier in Kapitel 2.2 vorgestellten Regeln gestaltet ist.

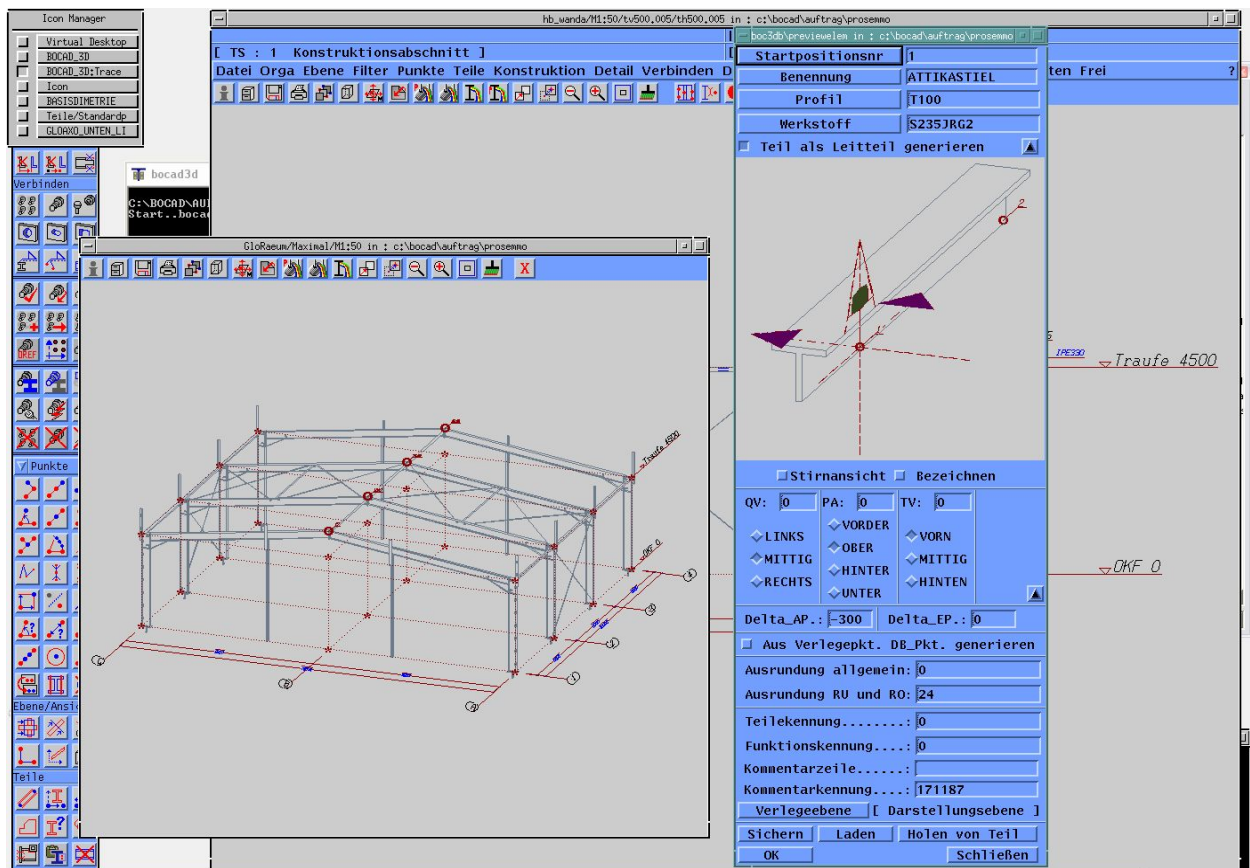


Abb. 31: Klassische Benutzungsoberfläche eines CAD-Hochleistungssystems
Quelle: BOCAD

Aufgrund der oben genannten Emulation erscheinen die einzelnen Fenster (Task) nicht in der Windows Taskleiste, sondern im sog. „Icon Manager“, einem eigenen Fenster auf dem Bildschirm. Fenster werden nicht durch einen Mausklick in die Fensterfläche in den Vordergrund geholt, sondern nur, wenn der Rand gewählt wird. Dies ist ein für Windows-Benutzer ungewohntes Vorgehen, nicht nur für Studierende sondern auch für Ingenieure im Bauwesen.

Ebenfalls aufgrund der Emulation erscheinen Schaltflächen und Kontrollelemente, wie z.B. Optionsfelder und Eingabefelder, nicht in der gewohnten Windowsdarstellung. Ein aktiviertes Kontrollkästchen erscheint schlecht erkennbar und missverständlich als „ein-

gedrückt“, während es in Windows klar und eindeutig durch ein Häkchen als markiert angezeigt wird (siehe Vergleich in Abb. 32).



Abb. 32: Vergleich Kontrollkästchen zwischen BOCAD-3D (links) und Windows (rechts)

Auch beim Ausfüllen von Dialogfeldern muss aufgrund der Emulation die Maus innerhalb des Dialogfensters verharren damit der Eingabefokus im Textfeld bleibt.

All dies sind Probleme, die auf die Emulation eines in der Baupraxis unüblichen Unix-Systems zurückzuführen sind. Die Betrachtung von BOCAD-3D in Version 20 wird hier Verbesserungen zeigen. BOCAD nähert sich so den üblichen Industriestandards, die Schulungsaufwand verringern.

Neben diesen durch die Emulation hervorgerufenen Problemen bei der Bedienung des Programms weicht BOCAD-3D auch in anderer Hinsicht von bekannten Verhaltensweisen bei Benutzungsoberflächen ab.

So folgt BOCAD-3D nicht dem im Styleguide empfohlenen Substantiv-Verb-Vorgehen [MS-01], sondern benutzt die nicht empfohlene Variante Verb-Substantiv. Zum Löschen eines konstruierten Objektes muss also zuerst der Befehl „Löschen“ (Verb) gewählt werden und danach das Element, welches gelöscht werden soll (Substantiv). Die üblichere Substantiv-Verb Vorgangsfolge wird einprägsam beschrieben mit „Erst markieren, dann operieren“.

Der Nachteil der Verb-Substantiv Variante ist, dass ein Modus aktiviert wird und der Benutzer sich den Befehl (z.B. Teile – Löschen) merken muss bis er das Objekt ausgewählt und somit den Befehl angewendet hat. Bei Ablenkung z.B. durch einen Anruf, eine E-Mail, Besucher etc. kann der Benutzer den Befehl vergessen haben und sich bestenfalls wundern, warum ein Objekt, das er anklickt, gelöscht wird.

Beim Substantiv-Verb-Vorgehen wählt der Benutzer erst das Objekt aus, das dann dauerhaft markiert aufleuchtet, und danach den Befehl, den er darauf anwenden möchte. Das System wird so nicht in einen Modus geschaltet, der zu unerwünschten Resultaten führt [Rask00].

Eine Effizienzsteigerung ergäbe sich, wenn der Benutzer über ein Kontextmenü nur die Befehle angezeigt bekommt, die aktuell auf dieses Objekt anwendbar sind. Diese Vorgehensweise, kombiniert mit international verständlichen Symbolen anstelle von Text, würde für weltweite Benutzbarkeit ohne Übersetzungsaufwand sorgen.



File Orga Ebene Filter Punkte Teile Konstruktion Maßen Grafik Zeichnung Listen Frei ?

Abb. 33: BOCAD-3D Hauptmenü (Ausschnitt)

Das Hauptmenü (Bild 33) bereitet Zuordnungsprobleme. So befindet sich der Befehl „Löschen“ in sämtlichen Menüs, also auch im Menü „Punkte“ wie im Menü „Teile“. Der Benutzer gibt das zu löschende Objekt (Punkt oder Teil) also nicht nur durch das Auswählen des Objektes vor, sondern er muss zudem im richtigen Menü den entsprechenden Befehl auswählen. Der Befehl „Löschen“ aus dem Menü „Punkte“ wird somit nur Punkte und keine Bauteile löschen.

Wie viel klarer und effizienter wäre es, stattdessen die zu löschenden Objekte auszuwählen und dann die weltweit bekannte Taste „ENTF“ („Delete“) zu drücken!

Bild 34 zeigt das umfangreiche, wenig strukturierte Menü „Teile“. In diesem findet der Benutzer nicht nur Befehle zum Erzeugen von Teilen, sondern auch zum Bearbeiten und Löschen. Dies ist zum einen ungewöhnlich, denn normalerweise finden sich Befehle zum Bearbeiten stets unter dem Menüpunkt „Bearbeiten“, so wie es im Styleguide [MS-01] vorgegeben wird.

Zum anderen ist die Struktur ungewöhnlich. Das Menü ist unterteilt in Abschnitte (Standardprofile, Sonderprofile, etc), die durch nicht auszuwählende Menüeinträge kombiniert mit Trennzeichen gebildet werden. Dies ist ein in dieser Form ungewohntes GUI-Element und wird daher von den in den Untersuchungen an der Bergischen Universität Wuppertal eingesetzten Benutzern intuitiv nicht verstanden. Gängiger wäre die Verwendung von sog. kaskadierenden Menüeinträgen.

Ebenso verstößt die Benennung der Menüpunkte zum Großteil gegen die Konvention, dass auf ein Substantiv im Menütitel stets Verben als Menüeinträge folgen sollten. So fehlt dem Befehl „Teile – 2-Punkte-Form“ das Verb „platzieren“. Die Menüs wirken so auf neue Anwender ungewohnt, also unverständlich.

Ähnlich undurchdacht zusammengestellt erscheinen die anderen Menüs von BOCAD-3D



Abb. 34: BOCAD-3D Menü "Teile"

Version 19.

Befragungen von Studierenden des ersten Semesters an der Bergischen Universität Wuppertal zeigen, dass die Hauptmenüs „Punkte“, „Teile“, „Konstruktion“ und „Detail“ für den Laien nicht verständlich sind. So mischen sich in allen Menüs, wie bereits für das Menü „Teile“ aufgezeigt, Befehle zum Erzeugen und Bearbeiten der jeweiligen Elemente. Hierbei ist es für die Studierenden nicht erkenntlich, wann ein Element als „Teil“ und wann es als „Konstruktionselement“ zu verstehen ist.

Statt unter „Bearbeiten“ finden sich unter „Detail“ weitere Befehle, mit denen Teile und Konstruktionselemente bearbeitet werden können. Sie können angepasst, verschnitten, kopiert sowie verschoben und gespiegelt werden. Allerdings lassen sich die Aktionen Kopieren, Verschieben und Spiegeln aber wiederum nicht auf Punkte anwenden.

Zwar wird ein fachspezifisches Programm wie BO-CAD-3D gelegentlich Sonderlösungen für die Benennung von Menüs finden müssen. Dann allerdings sollten diese logisch und durchgängig gewählt werden, d.h. alle Befehle, mit denen Objekte – gleich weder Art – bearbeitet werden können, müssen im Menü „Bearbeiten“ zu finden sein. Menübefehle, die neue Bauteile konstruieren, gehören in ein zentrales Menü „Teile konstruieren“. Dort sind dann, wie in Vorgaben des Microsoft Styleguides, alle zu konstruierenden Teile als Substantive gelistet.

Auch ein Kontextmenü, welches nach Rechtsklick auf ein Objekt nur die Befehle anzeigt, die in diesem Kontext auf das Objekt anwendbar sind, würde

die Effizienz steigern. Bei Auswahl eines Punktes wären dann nur Befehle zu zeigen, die auf einen Punkt anwendbar sind, bei Auswahl eines Bauteils nur die hierfür gültigen Befehle.

Es bleibt also festzuhalten, dass auch bei einer textverwendenden Benutzungsoberfläche ein grammatikalisch konsequenter, systematischer Aufbau die Effizienz der Benutzer steigert und den Lernaufwand senkt.

Wesentliche Voraussetzung für fast alle hier vorgeschlagenen Vorgehensweisen ist es aber, dass die Objekte direkt auszuwählen sind, um die Objekte mit direkter Manipulation statt über mehrschrittige Umwege zu bearbeiten. Dies ist in BOCAD-3D weitgehend nicht der Fall. Was unter direkter Auswahl und Bearbeitung zu verstehen ist, zeigt das folgende Beispiel, Bild 35.

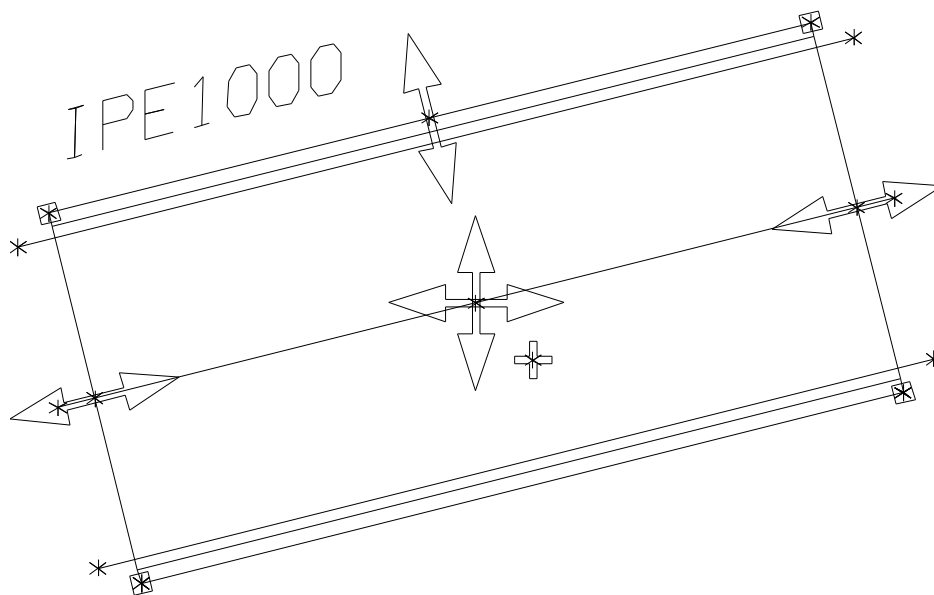


Abb. 35: Anfasser für Verlängern, Profiländern, Verschieben, Kopieren, Quelle: [Chan02]

Nach Auswahl eines Elementes wird dieses elementspezifisch durch sog. Anfasser als markiert angezeigt. Dabei ist der Begriff „Anfasser“, ebenso wie der englische Begriff „Handles“, nicht formal definiert. Er hat sich aber für diese Griffe, die direkte Manipulation von Objekten ermöglichen, in der Fachliteratur etabliert.

Mithilfe dieser Anfasser kann das Objekt verschoben, kopiert bzw. in der Profilgröße oder in der Länge verändert werden – ein veränderter Mauscursor zeigt dem Benutzer den gerade ausgeführten Befehl an. Einen Entwurf eines solchen international sprachfrei verständlichen Ansatzes zeigt Abb. 35.

Ein Doppelklick auf das Element öffnet ein Dialogfenster, in dem die für dieses Objekt sinnvollen Informationen angezeigt und editiert werden können. Informationen, die auf dieses Element nicht angewendet werden können, werden dem Benutzer nicht angezeigt.

Diese Vorgehensweise ist für alle Objekte möglich, ein Teil, ein Punkt oder ein Konstruktionselement. Der Benutzer muss nicht mehr die Unterscheidung treffen, sondern das System erkennt das Objekt und zeigt die in einer Datenbank gespeicherten Informationen entsprechend an.

Das Bauingenieurwesen bietet als zeichnungsbasiertes Fachgebiet einen reichhaltigen Fundus, um in einer solchen Benutzungsoberfläche weitgehend auf Text verzichten zu können. Objekte können durch die Anfasser direkt skaliert werden, wobei die Endpunkte mittels Objektfang automatisch an anderen Objekten oder Punkten ausgerichtet werden (siehe CorelDraw 12, Bild 23). Eine Profiländerung ist über Ziehen quer am Profilstab oder über Auswahlfelder möglich, wobei die Profildaten aus einer Profiltabelle eingelesen werden.

In Dialogfenstern und Symbolleisten bietet sich durch Symbole für Schweißnähte, Schraubensymbole sowie Maßketten und weitere international geläufige Symbole die Möglichkeit, möglichst weitgehend auf Texte zu verzichten (siehe Symbolleisten bei CorelDraw 12, Bild 26).

Die in Kapitel 5 gezeigten Software-Prototypen zeigen den in dieser Arbeit entwickelten programmtechnischen Ansatz, der ein Re-Engineering von CAD-Hochleistungssystemen gestattet, um sie für die Zukunftsmärkte hinreichend effizient und verständlich zu gestalten. Zudem gibt es auch positive Erkenntnisse aus dieser Betrachtung:

Ein Idealfall im Sinne dieser Arbeit sind die in BOCAD-3D Version 19 implementierten Leitbilder als eine Form der Dialogfenster, die Dokumentation, Hilfe und grafisches Eingabeformular vereint, siehe Bild 36.

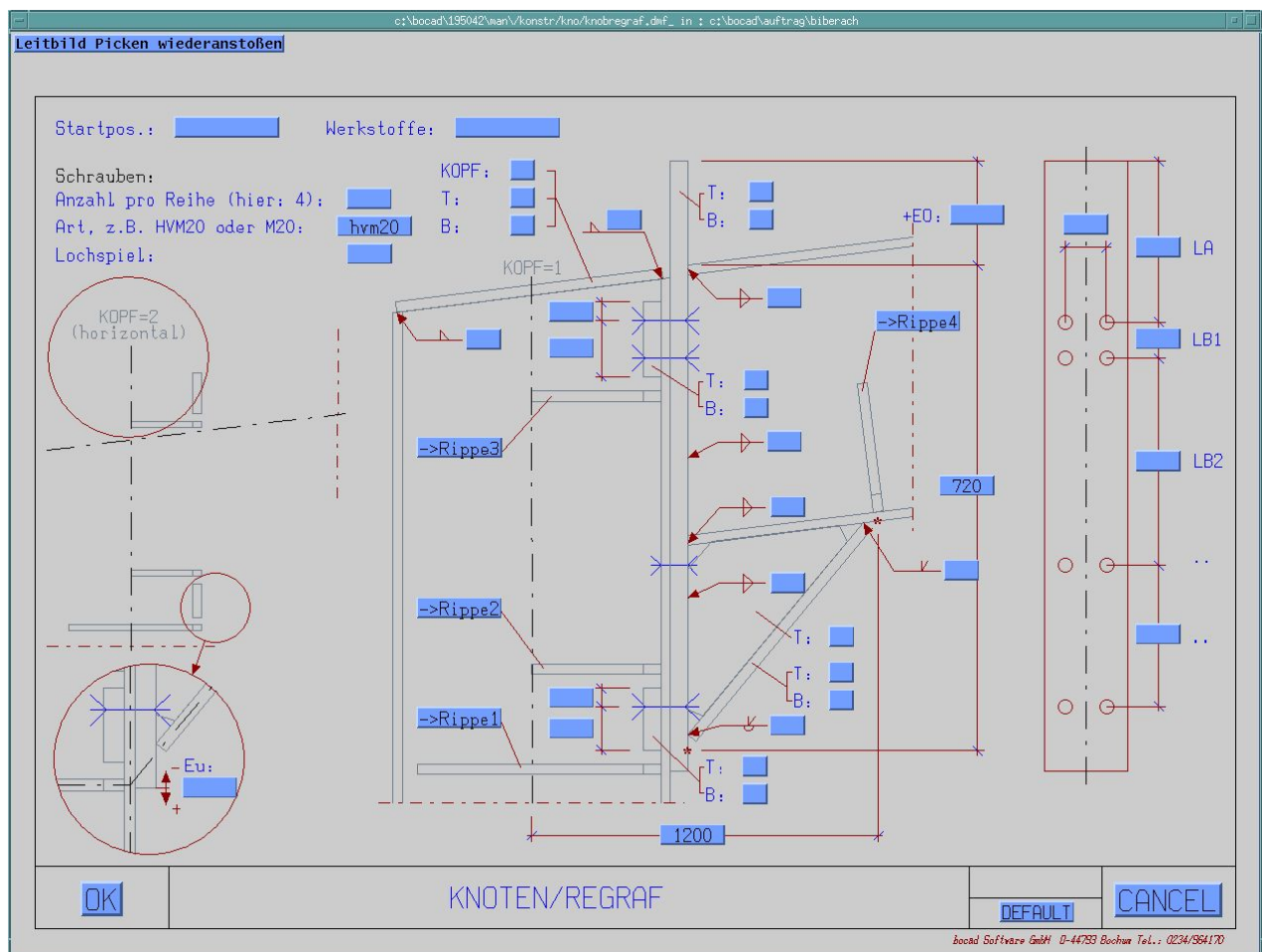


Abb. 36: BOCAD-3D Version 19, Leitbild zur Konstruktion einer Rahmenecke Typ "Grafbau"

Bild 36 zeigt ein Leitbild zur Konstruktion von Rahmenecken des Typs „Grafbau“. Der Benutzer erhält ein Dialogfenster, welches in unmaßstäblich gut lesbarer Form ein zeichnerisches Muster des zu konstruierenden Elements anzeigt. Darin kann der Benutzer die gewünschten Spezifikationen des zu konstruierenden Elements über Eingabefelder, Drehknöpfe oder Optionsschaltflächen direkt im Leitbild eintragen. Hat er die Hauptspezifikationen festgelegt, werden daraus alle Folgespezifikationen automatisch über die entsprechende CAD-Konstruktionsmethode errechnet.

Verbessert um die bisher vorgestellten Gestaltungsregeln sind Leitbilder ein Musterfall für das anzustrebende Ziel einer intuitiven, textfreien Benutzungsoberfläche, wie sie in [Azad07] und [Weck07] konzipiert wird. Diese Arbeit zeigt in Kapitel 5 den entsprechenden programmtechnischen Ansatz dazu.

2.3.6 BOCAD-3D Version 20

Im Folgenden werden nur wesentliche Aspekte, die sich von Version 19 unterscheiden, untersucht, analysiert und Schlussfolgerungen hinsichtlich einer intuitiven und internationalen Benutzungsoberfläche abgeleitet.

In Version 20 von BOCAD-3D entfällt die Emulation eines Unix-Systems auf einem Windows-Rechner. BOCAD-3D erscheint nunmehr wie ein Windows-Programm mit allen Verhaltensweisen, die der Benutzer gewohnt ist (Bild 37). Schaltflächen und Kontrollelemente entsprechen dem Windows-Standard und auch die Handhabung der Fenster ist Windows-konform.

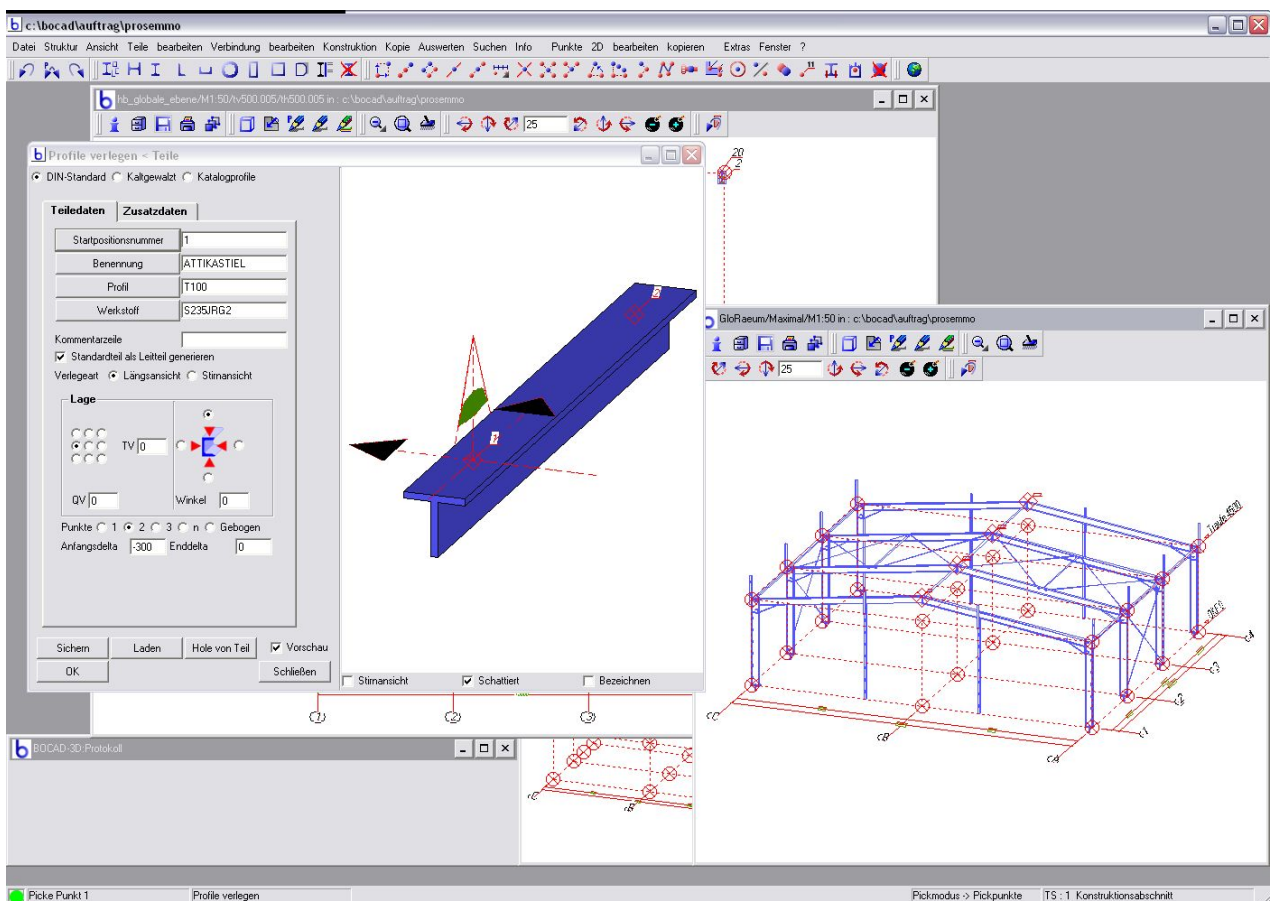


Abb. 37: Benutzungsoberfläche des CAD-Hochleistungssystems BOCAD-3D, Version 20

Dies wurde erreicht, indem die Benutzungsoberfläche (statt mit der unter Version 19 genutzten Emulation durch die Exceed-Software) mit Qt gestaltet wurde. Die plattformüber-

greifende und programmiersprachenunabhängige Qt-Programmbibliothek, die auch im weiteren Verlauf dieser Arbeit für die Erstellung einiger Software-Prototypen verwendet wird, ist sowohl für die Betriebssysteme Windows als auch für Linux/Unix verfügbar, so dass auch die aktuelle Version von BOCAD-3D weiterhin auf beiden Systemen einzusetzen ist.

Neu ist die Verwendung einer Statuszeile am Fuß des Anwendungsfensters. Obwohl auch das Hauptmenü neu gestaltet wurde, blieben die im vorigen Kapitel analysierten, unsystematischen Ansätze unkorrigiert mit allen negativen Folgen für die Benutzungsoberfläche.



Abb. 38: BOCAD-3D Hauptmenü in Version 20

Bild 38 zeigt das Hauptmenü. Wesentliche Änderungen betreffen die Bezeichnung der Menüs. Abb. 39 zeigt das neugestaltete Menü „Teile“, das durch weniger Menübefehle und windows-konforme Gestaltung zumindest übersichtlicher ist.



Abb. 39: Neues Hauptmenü "Teile"

Durch ein verbessertes Verfahren für das Platzieren von Stäben und Bauteilen kann nunmehr auf die Trennung in 2-Punkte, 3-Punkte oder n-Punkte-Formen verzichtet werden. So findet sich nur noch der Befehl „Profile verlegen“ (der korrekter „Profilstäbe platzieren“ lauten sollte), der die früheren Befehle ersetzt. Auch die Unterteilung in Standard- und Sonderprofile konnte so aufgehoben werden.

Befehle zum Bearbeiten von Teilen wurden aus dem Menü „Teile“ entfernt und in eigenes Menü „bearbeiten“ überführt. Dies steigert die Übersichtlichkeit beider Menüs nennenswert.

Allerdings enthält auch dieses Menü einen Sonderweg: Betitelte Abschnitte, hier der Abschnitt „DaWa“, sind im Windows-Styleguide nicht vorgesehen. Da Dach- und Wandkonstruktion

nen keine Profilstäbe sind, wäre hier ein eigenes Menü „DaWa“, wie in Version 19, verständlicher und konsequenter strukturiert.

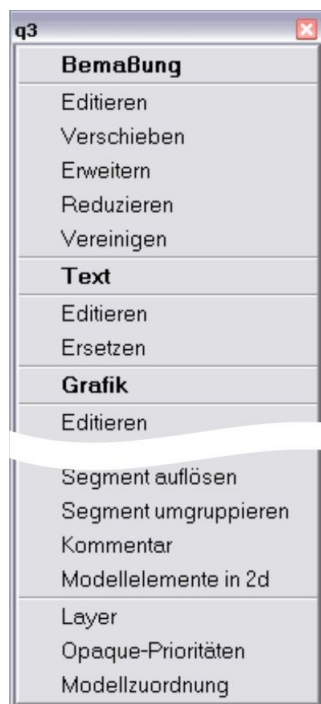


Abb. 40: Menü "bearbeiten" von 2D-Elementen

Wie bereits erwähnt, sind nun sämtliche Befehle zum Bearbeiten der Teile konsequent in einem eigenen Menü „bearbeiten“ zusammengefasst. In diesem befinden sich somit auch alle Befehle aus dem früheren Menü „Detail“.

Nach wie vor aber gibt es in Version 20 statt einem einzigen insgesamt drei Menüs „bearbeiten“, unkonventionellerweise in Kleinbuchstaben (siehe Abb. 38), zum Bearbeiten von Teilen, Verbindungen und „2D“-Elementen (Bemaßungen, Texte und Grafikelemente).

Dieser Mangel ist eine Konsequenz des unüblichen Verb-Substantiv-Verfahrens, das auch in Version 20 beibehalten wurde. Im üblichen Substantiv-Verb-Verfahren würde beispielsweise das Menü „2D bearbeiten“ (siehe Bild 40) überflüssig. Ein Grafikelement (Linie, Kreis etc.) könnte nämlich dann direkt ausgewählt und danach mit der Maus verschoben oder mit Doppelklick in seinen Eigenschaften verändert werden. Ebenso verhielte es sich mit dem Menü „Verbindungen bearbeiten“ zur Manipulation von Schweißnähten, Schraubenbildern und sonstige Verbindungen.

Einzig das Menü „Teile bearbeiten“ bliebe, in dem sich dann nur noch fachspezifische Befehle zum Detaillieren (z.B. Verschneiden, Abschneiden etc.) befänden.

Eine weitere Chance zur Verständlichkeit der Benutzungsoberfläche wurde vertan. Bild 37 zeigte die mit graphischen Symbolen reichhaltig ausgestattete Benutzungsoberfläche der Version 20.

Den vielen Menübefehlen jedoch wurden keine Symbole zugeordnet, siehe Abb. 40. Wie eine Menüleiste effizient und verständlich zu gestalten wäre, zeigt der Vorschlag in Bild 41. Er enthält alle Merkmale, die zu elementaren Bestandteilen moderner Benutzungsoberflächen geworden sind. Die Verwendung von Symbolen vor jedem Befehl der Liste hilft, dem Benutzer die Bedeutung der Symbole einzuprägen. Er erlernt so über die Ver-

wendung der Menüs die Bedeutung der Symbole und kann diese später dann effizient in den Symbolleisten erkennen und nutzen.

Der programmtechnische Ansatz in Kapitel 5 wird anhand eines Software-Prototypen aufzeigen, wie ein solches, intuitiv verständliches und zeitgemäßen Anforderungen genügendes Menü zu programmieren ist. Die auch in BOCAD-3D Version 20 verwendete Klassenbibliothek Qt bietet dem Entwickler hierfür ein innovatives Konzept, das BOCAD nicht erkannt bzw. nicht genutzt hat.



Abb. 41: Gestaltung einer effizienten und verständlichen Menüleiste

Auch BOCAD-3D Version 20 bietet dem Benutzer bei Auswahl einer automatischen Konstruktionsmethode die aus Version 19 bekannten Leitbilder.

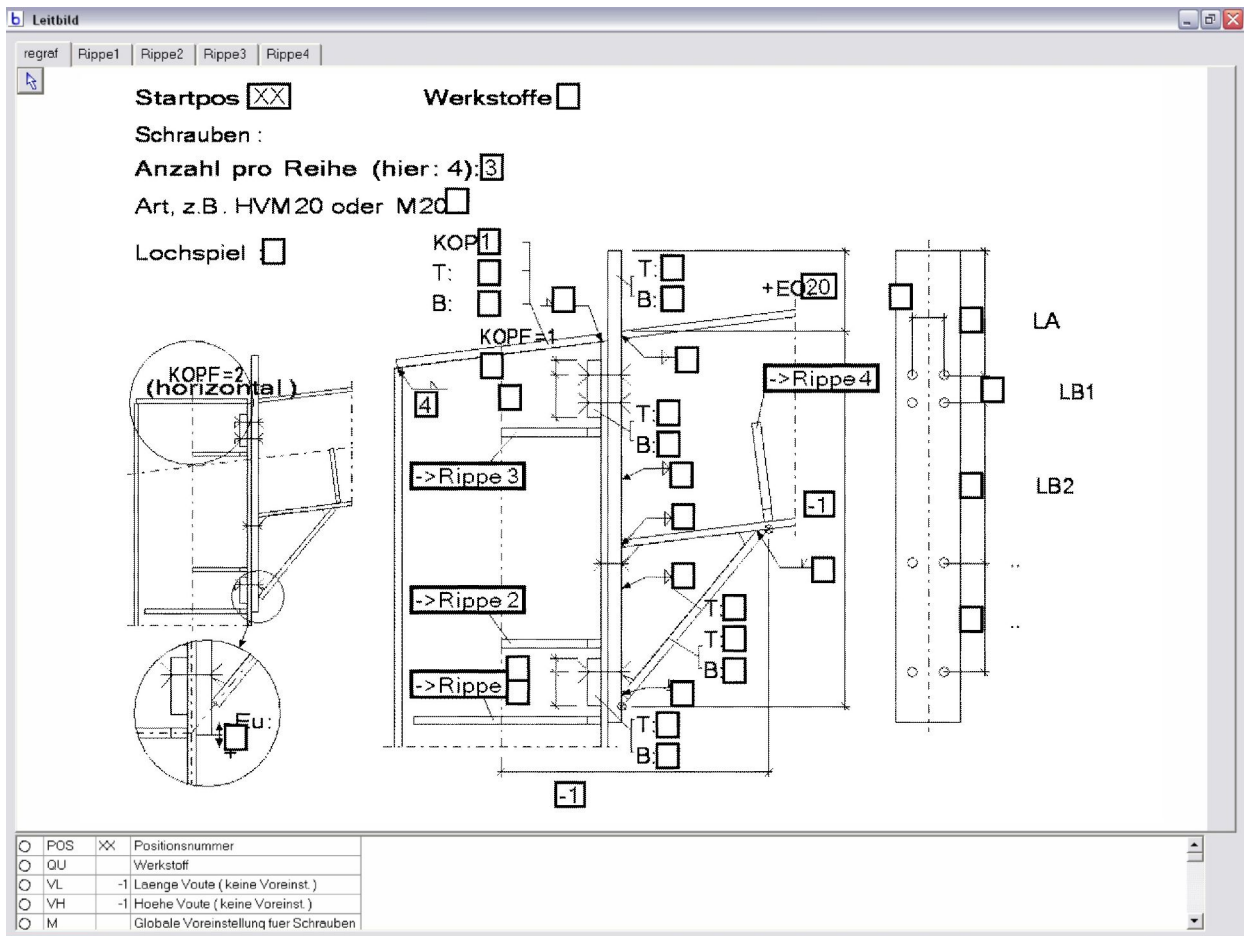


Abb. 42: BOCAD-3D Version 20, Leitbild Rahmenecke Typ "Grafbau"

Doch in Version 20 ist diese grundlegend innovative Lösung der intuitiven Benutzerführung nicht mehr in der aus Version 19 bekannten Form umgesetzt, wie Bild 42 zeigt. Das wesentliche Merkmal, Benutzereingaben direkt in der Zeichnung auf dem Bildschirm vornehmen zu können, ist nicht mehr möglich. Benutzereingaben erfolgen in dem zweigeteilten Dialogfenster im unteren, kleineren Teil des Fensters in einer Tabelle. Dabei wird die BOCAD-interne Variablenbezeichnung sowie ein deutscher Beschreibungstext ausgegeben. In der dritten Tabellenspalte kann der Benutzer seine Spezifikationen eingeben.

Ein direktes Bearbeiten der Eingaben im Leitbild ist nicht mehr möglich. Diese Form der Leitbilder widerspricht dem innovativen Ansatz aus BOCAD-3D Version 19 und dem Bestreben, eine intuitiv verständliche, sprachfreie Benutzungsoberfläche zu schaffen, komplett. Den Entwicklern ist es nicht gelungen, einen geeigneten programmtechnischen Ansatz zu entwickeln.

2.3.7 Tekla Structures 11.2

Der Vergleich der Benutzungsoberflächen beider im Weltmarkt konkurrierenden CAD-Hochleistungssysteme für Stahlbau lässt interessante Erkenntnisse zum Thema dieser Arbeit erwarten.

Auch bei Tekla Structures, hervorgegangen aus Tekla Xsteel und hier in Version 11.2 vorliegend, handelt es sich um ein 3D-CAD-Hochleistungssystem ähnlich zu BOCAD-3D.

Hier werden ebenso wie bei BOCAD-3D im räumlichen Modell Profilstäbe platziert und CAD-Konstruktionsmethoden angewandt, um so das komplette Bauwerk am PC zu modellieren und anschließend automatisch sämtliche technischen Unterlagen erzeugen zu lassen.

Basierend auf einem sehr ähnlichen 3D-Modell wurde die Benutzungsoberfläche von Tekla Structures wesentlich weiter entwickelt als die von BOCAD-3D, siehe Bild 43.

Tekla Structures besitzt einige Windows-Konformität bezüglich der Bedienung der Fenster, Dialogfenster, Menüs und Symbolleisten. Dies wird durch die Objektorientierung aller Elemente auf dem Bildschirm ergänzt, d.h. alle Elemente sind durch Mausklick auswähl-

und editierbar. Allerdings sind die Objekte nicht interaktiv direkt über Anfasser zu bearbeiten, z.B. das Verändern der Profillänge oder das interaktive Verschieben. Das Kontextmenü bietet nur wenige Standardbefehle an, die wiederum Texteingaben erforderlich machen.

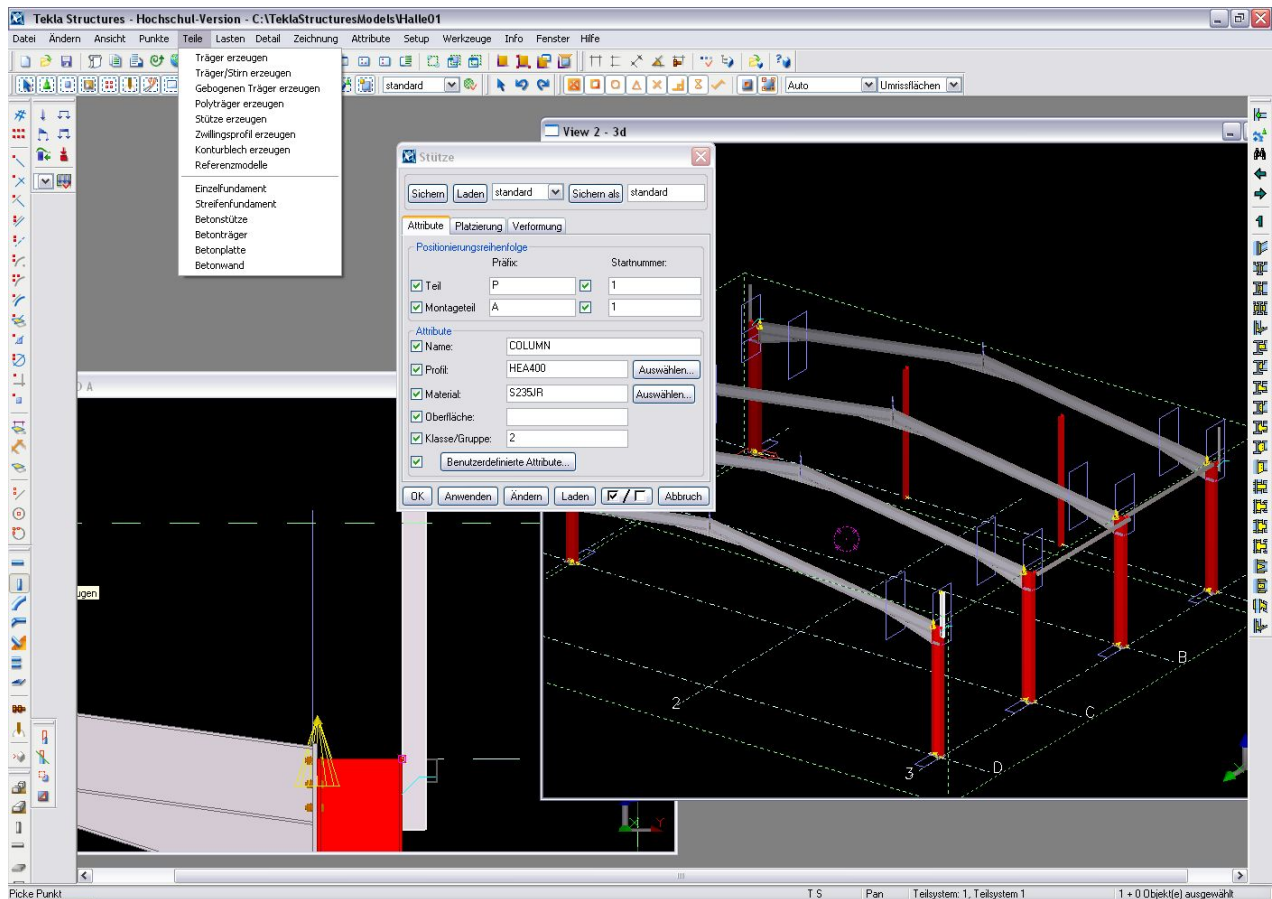


Abb. 43: Tekla Structures 11.2

Eine intuitive Benutzungsoberfläche für die direkte Manipulation ohne Texte wie in Abb. 35 vorgeschlagen, auf der Elemente durch Anfasser in ihrer Länge oder Lage verändert oder Profile ausgetauscht werden können, wurde somit auch hier nicht verwirklicht. Durch den Systemansatz zur Objektauswahl wäre dies möglich gewesen.

Symbole in den Menüleisten fehlen. Auch Umfang und Anzahl der Symbolleisten stehen einer intuitiven Benutzung entgegen.

Beim Benutzer prägen sich daher zwei Verhaltensmuster aus, die sich eigentlich gegenseitig unterstützen sollten, hier aber gegeneinander wirken: Hat der Anwender nach lan-

ger Schulung erst einmal alle Befehle in den Menüs gefunden, wird er nicht noch einmal die Bedeutung der Symbole erkunden wollen. Er wird fortan mit den Textmenüs arbeiten und die Symbole aufgrund fehlender Lernhilfen nicht nutzen. Entsprechende Erfahrungen in der Lehre an der Bergischen Universität Wuppertal wurden mit Mitarbeitern und Studierenden gemacht.

Weitere Sonderwege der von Tekla konzipierten Benutzungsoberfläche sind nicht systemkonform und erschweren eine intuitive Nutzung.

So verfügen die Dialogfenster (Bild 44, Dialog zum Einfügen einer Stütze) über Schaltflächen, deren Wirkungsweise sich nicht unmittelbar erschließt. Änderungen der Eingaben werden nur dann in die Befehlsausführung übernommen, wenn die Schaltfläche „Anwenden“ gedrückt wird. Dabei bleibt das Fenster weiterhin geöffnet. Ein Klick auf „OK“ schließt den Dialog, übernimmt aber nicht wie üblich die im Fenster getroffenen Einstellungen.

Ebenso ungewöhnlich ist die Bedeutung der Schaltfläche „Ändern“: Wird ein bereits konstruiertes Teil durch Doppelklick gewählt, öffnet sich ein Dialog wie in Abb. 44. Änderungen am ausgewählten Objekt werden aber nur übernommen, wenn dann zusätzlich auf die Schaltfläche „Ändern“ geklickt wird. Der Dialog bleibt danach auch hier weiter offen. Erst mit „OK“ wird er geschlossen.

Um eingegebene Änderungen zu verwerfen, ist international die Schaltfläche „Abbrechen“ üblich, die es auch bei Tekla Structures gibt. Die Schaltfläche „OK“ ebenso wirken zu lassen, ist sinnwidrig, unkonventionell und nicht den Styleguides und den Forderungen der EN-ISO entsprechend. Der Benutzer erwartet, dass seine Eingaben mit Klick auf „OK“ übernommen und das Dialogfenster geschlossen wird. Die Unterscheidung, ob Eingaben auf ein neues oder ein vorhandenes Objekt angewendet werden (also das Objekt geändert werden soll), verwirrt den Anwender vollends.

Nachvollziehbare Gründe für den geschilderten Sonderweg waren nicht zu erhalten.



Abb. 44: Tekla Structures Dialogfenster „Stütze“

Daneben fehlt die Definition einer „Standard-Schaltfläche“ (*Default-Button*). Die mit dieser Schaltfläche verbundene Aktion wird dann ausgeführt, wenn der Benutzer die Eingabetaste („Return-Taste“) auf seiner Tastatur drückt. Normalerweise wird die „OK“-Schaltfläche als Standard definiert, so dass nach Betätigung der Eingabetaste die vom Benutzer eingegebenen Werte übernommen und das Dialogfenster geschlossen wird. Dies ist ein elementarer Teil einer jeden Dialogbox und als solcher auch in [MS-01] gefordert.

Die Dialogfelder in Tekla Structures sind lediglich textbasiert. Hier ist der entsprechende Dialog von BOCAD-3D hervorzuheben, der, speziell in Version 20, weitaus sprachfreier und intuitiver ist (siehe Bild 37). Vorallem das Fehlen einer „Onlinevorschau“, also einer Vorschau des Teils im Dialogfenster, ist effizienzmindernd.

Ebenso finden sich auch in Tekla Structures, ähnlich zu Leitbildern in BOCAD-3D, Dialogfenster zur Spezifikation von komplexen Konstruktionsmakros, siehe Bild 45. Man er-

kennt, dass die Dialogfenster in mehrere Ebenen aufgeteilt sind, die über Reiterlaschen zu erreichen sind.

Im linken Teil der Abb. 45 erkennt man die erste Seite des Dialogfensters zur Konstruktion einer Stirnplatte in Tekla Structures. Ähnlich zu BOCAD-3D Version 19 gibt der Benutzer auch hier über eine Zeichnung und entsprechende Eingabefelder seine Spezifikationen ein.

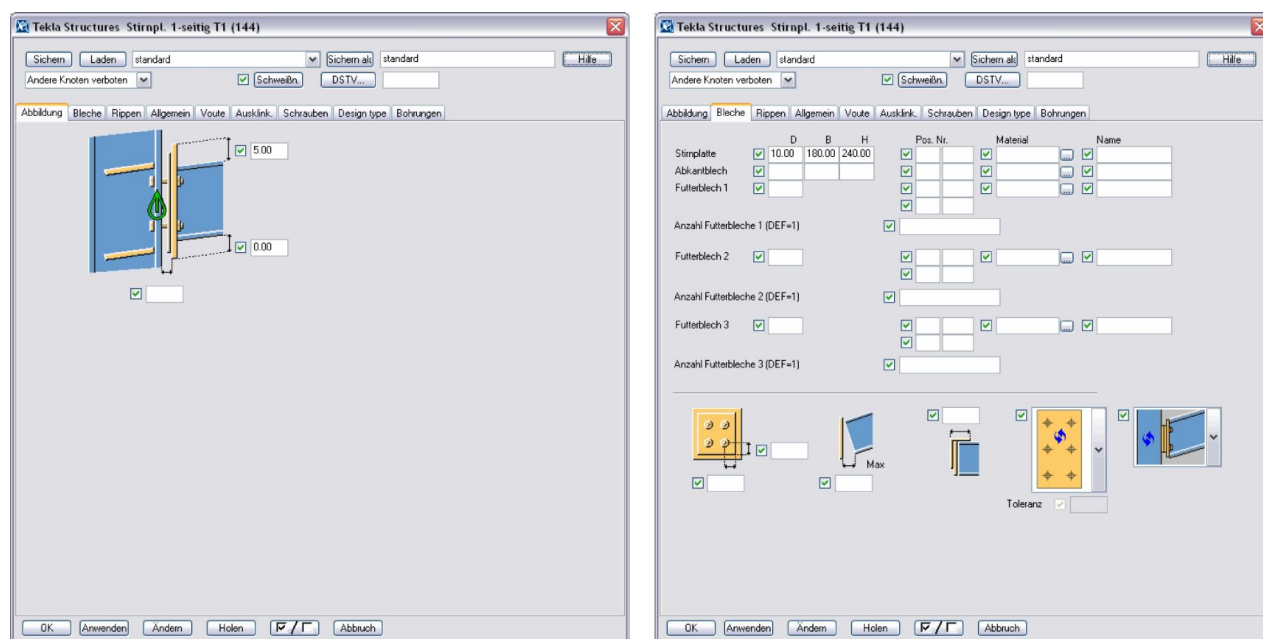


Abb. 45: Dialogfenster zur Konstruktion einer Stirnplatte in Tekla Structures

Der rechte Teil der Abb. 45 zeigt hingegen, dass die weiteren Spezifikationen nicht über intuitiv zu verstehende Zeichnungssymbole eingegeben werden. Neben der deutschen Beschriftung der Reiter erfolgt auch die Bezeichnung der Eingabefelder in Textform. Dies widerspricht nicht nur einer von Landessprachen unabhängigen Benutzungsoberfläche, der Benutzer wird zudem auch Probleme bei der Zuordnung seiner Werte in die Eingabefelder haben, da die Beschriftungen nicht eindeutig genug sind.

Leitbilder entsprechend BOCAD-3D Version 19, wie in Bild 36 dargestellt, sind für Effizienz und Verständlichkeit daher die beste aller hier verglichenen Lösungen.

So greift auch Tekla Structures nicht den innovativen Ansatz aus BOCAD-3D Version 19 auf, mittels direkter Eingabe in eine intuitiv verständliche Bauzeichnung eine sprachfreie Benutzungsoberfläche zu schaffen.

Darüber hinaus stehen für Träger, Stützen, gebogene Träger und Polyträger jeweils eigene Symbole zur Auswahl, obwohl das dahinterstehende Dialogfenster stets dasselbe ist. Auch hier zeigt BOCAD-3D in Version 20 mit nur noch einem Befehl „Teile“ eine intuitivere Vorgehensweise.

Die Benutzungsoberfläche schafft weitere Erschwernis durch verwirrende Modi der Symbole in den Symbolleisten.

So aktiviert ein Mausklick auf ein Symbol diesen Befehl, schaltet somit in den spezifischen Modus. Für Eingaben mit entsprechendem Öffnen des Dialogfensters ist aber ein Doppelklick auf das Symbol notwendig. Durch einen Doppelklick nach bereits erfolgten einfachen Klick auf das Symbol wird das Dialogfenster zwar geöffnet, aber der Benutzer befindet sich nun nicht mehr im spezifischen Modus, obwohl er das entsprechende Dialogfenster angezeigt bekommt. Der Doppelklick deaktiviert den Modus also wieder.

Einzig ein direkter Doppelklick wirkt erwartungskonform, d.h. er aktiviert den Modus und öffnet das Dialogfenster.

Tekla Structures kann hier als negatives Musterbeispiel für die Forderung von Raskin in [Rask00] gelten, nachdem Modi in Benutzungsoberflächen zu vermeiden sind.

Auch hier gibt BOCAD-3D das bessere Beispiel: Das Dialogfenster zum Verändern des Befehls kann über die rechte Maustaste jederzeit geöffnet werden, ohne das System in einen Modus zu versetzen (oder aus diesem Modus heraus). In einem System mit Kontextmenüs, wie Tekla Structures, ist ein Klick mit der mittleren Maustaste eine entsprechende Lösung.

Tekla Structures bietet aber auch innovative Ansätze. Das freie Drehen des Gebäudemodells im Raum, das unendlich tiefe Undo sowie die Möglichkeit, jeden Konstruktionschritt und jede angewandte Konstruktionsmethode auch zu einem späteren Zeitpunkt durch Doppelklick editieren zu können, sind Effizienz steigernd. Möglich macht dies nicht nur die native Unterstützung der 3D-Grafikengine von Windows, sondern auch eine Datenbank, in der diese Informationen platzsparend abgelegt werden.

3 Konzeption „direkter“, sprach- und textfreier Benutzungsoberflächen

3.1 Grundlagen

Raskin, Entwickler der grafischen Benutzungsoberfläche des Apple Macintosh Computers, nennt in [Rask00] zahlreiche Ansätze zur Gestaltung und Bewertung moderner und intuitiver Benutzungsoberflächen:

„Eine Oberfläche zu entwerfen ist ein bisschen, wie ein Haus zu bauen: Wenn das Fundament nicht solide ist, kann keine noch so schöne Gestaltung diesen strukturellen Schaden wieder beheben“ so Jef Raskin. Die Gestaltung der Benutzungsoberfläche ist so nicht nur Teil des Pflichtenheftes jeder Softwareentwicklung, sondern einzige beiderseits verständliche Schnittstelle zwischen Anwendern und Programmierern.

Modi wie bei Tekla Structures erschweren den Überblick über das Verhalten von Benutzungsoberflächen. Zum Verständnis ist zunächst die Definition der Geste notwendig: Eine Geste ist „die Folge an Handlungen, die abläuft, sobald sie in Gang gesetzt wurde“ [Rask00]. Übliche Benutzungsschnittstellen sehen verschiedene Bedeutungen derselben Geste vor. So fügt in Windows das Drücken der Eingabetaste in einem Texteditor einen Zeilenumbruch ein, in einem Dialogfenster bestätigt dieselbe Geste die Eingabe und in einer Eingabeaufforderung wird der zuvor eingegebenen Befehl ausgeführt.

Der Modus legt in diesem Zusammenhang fest, wie das System auf eine Geste antwortet. Ein Umschalten des Modus in einem System führt also dazu, dass das Programm auf dieselbe Geste anders antwortet.

Das in Kapitel 2.3.7 vorgestellte CAD-System Tekla Structures verdeutlicht unfreiwillig, wie Modi nicht eingesetzt werden sollten. So wird der Modus, in dem sich das System befindet (aktiver Befehl = aktives Symbol) weder durch einen veränderten Cursor, noch durch ein verändertes Symbol angezeigt. Zwar werden unterschiedliche Modi genutzt, der jeweilige Zustand dem Benutzer aber vorenthalten.

Textverarbeitungsprogramme wie Word oder einfache Editoren wie Wordpad verfügen über einen weltweit bekannten Modus: „Einfügen“ bzw. „Überschreiben“.



Abb. 46: Modus "Einfügen" oder "Überschreiben"

Durch Drücken der Taste „Einfg“ auf der Tastatur wird der Modus umgeschaltet und in der Statuszeile der Anwendung angezeigt (siehe Abb. 46). Wie langjährige eigene Erfahrungen mit Studierenden zeigen, führt das versehentliche Drücken der Taste „Einfg“ auf der Tastatur zum ungewollten Löschen (Überschreiben) von Texten, da die Anzeige des Modus weit außerhalb des Zentrums der Aufmerksamkeit erfolgt. Die Studierenden verlieren wiederholt viel Zeit damit, diese unbeabsichtigte Umschaltung selbstständig rückgängig zu machen.

Für den Entwurf einer intuitiven Benutzungsoberfläche gilt es zudem, den Begriff der Sichtbarkeit zu definieren. So wird eine Funktion nur dann als sichtbar bezeichnet, wenn sie sich akut vom Benutzer erfassen lässt oder so kurz zuvor erfasst wurde, dass sie sich noch im Kurzzeitgedächtnis befindet [Rask00].

Auch eine Funktion, die der Benutzer nur nach längerer Suche findet oder unter Inanspruchnahme der Hilfe, gilt als unsichtbar.

Es ist erstrebenswert, dass jede Funktion für fachkundige Menschen direkt, also durch bloßen Anblick, erkennbar ist. Eine solche Funktion ist nach [Rask00] eindeutig.

So sind Drehknöpfe, Optionsfelder und Eingabefelder eindeutig, da sich ihr Sinn unmittelbar erschließt: Drehknöpfe können gedreht werden und so den Wert ändern, Optionsfelder sind auswählbar, ihr gewählter Status ist durch ein Häkchen eindeutig. In Eingabefelder sind eindeutig Texte einzutragen.

Beim Entwurf von Symbolen gilt es daher, Sichtbarkeit und Eindeutigkeit zu erzielen. Im CAD-System Tekla Structures ist dies nicht gelungen. Die viel zu große Anhäufung von Symbolen am Bildschirmrand verhindert die Sichtbarkeit im vorgenannten Sinn (siehe Bild 43).

Dort, wo es im weltweiten Straßenverkehr zwingend notwendig ist, gibt es bereits zahlreiche Symbole, die kulturübergreifend genutzt und verstanden werden: Verkehrszeichen bieten stark abstrahierte, eindeutige Symbole. Lediglich durch Form und Farbe werden sie sprachübergreifend sofort verstanden.



Abb. 47: Kulturübergreifendes Symbol, Quelle: [Hüb05]

Bestes Beispiel eines trotz lokaler Sprachverwendung international verständlichen Symbols ist das „Stopp“-Zeichen. Aufgrund von Form und Farbe (rot signalisiert bei Verkehrszeichen generell Verbot und Gefahr) ist es, selbst bei vollkommen unbekanntem Sprachtext, sofort verständlich (siehe Bild 47).



Abb. 48: Internationale Sportpiktogramme, Quelle: [Hüb05]

Auch manche Piktogramme zur Orientierung und Information sind derart prägnant gestaltet, dass sie international sprachfrei verstanden werden. Als Beispiel seien hier die Icons für Sportarten erwähnt, wie sie für die Olympischen Spiele in Berlin 1936 eingeführt und bei den Olympischen Spielen in London 1948 fortgeführt wurden [Hüb05].

Bild 48 zeigt, dass diese derartig abstrahiert sind, dass sie trotz individueller Gestaltung bei den verschiedenen Olympischen Spielen direkt erkannt werden.

Als Beispiel internationaler Verständlichkeit standen sie Pate für Orientierungssysteme auf Bahnhöfen, Flughäfen und an öffentlichen Orten.

Ein bekanntes Hinweisschild ist der „Fluchtweg“. International verständlich findet es sich in der uns bekannten Form in vielen Ländern. Bild 49 zeigt ein Beispiel in Spanien, dessen Textverzicht notwendig wäre.

In Bild 50 wird allerdings deutlich, wie wichtig die korrekte und durchdachte Gestaltung ist. So ist das Design dieses speziellen Schildes verfehlt. Es zeigt den Fliehenden in falscher Richtung an. Symbole müssen also durch Form, Farbe, Übersichtlichkeit und korrekte, den Sinn des Symbols unterstützende Elemente intuitiv verständlich sein.



Abb. 49: Internationaler Fluchtweghinweis
Quelle: [Hübn05]



Abb. 50: Richtungsgebundene Piktogramme, Quelle: [Hübn05]

Dies zeigt, wie standardisierte Symbole unser alltägliches Leben bereits jetzt durchdrungen haben und dass bei Software für Baufachleute sicher ein noch größeres Potential auszuschöpfen ist. Speziell für den wirtschaftlichen Vertrieb von Bausoftware muss dies auch bei der Gestaltung von grafischen Benutzungsoberflächen gelingen. So ist zwar die Übersetzung und Anpassung einer millionenfach verkauften Software wirtschaftlich lohnend, aber bei spezieller Branchensoftware mit wesentlich geringeren Stückzahlen ist dies nicht möglich. Sprach- und Texthürden wirken prohibitiv und auch das „Einigen“ auf eine Sprache (z.B. englisch) ist nicht realistisch, wenn man die Erfahrungen aus Entwicklungsländern einbezieht [Azad07].

Neben den bereits etablierten Zeichnungssymbolen ist vor allem die direkte Manipulation von Objekten elementarer Bestandteil einer solchen intuitiven, wirtschaftlichen Benutzungsoberfläche für das Bauwesen.

Die standardisierten Bauzeichnungen und Darstellungen bieten ein großes Potential, internationale Icons und Symbole zu entwickeln, die es ermöglichen, auf Landessprachen und Schriftzeichen in der Benutzungsoberfläche zu verzichten. Die Funktion „Teile verschweißen“ lässt sich z.B. durch das Symbol einer Schweißnaht darstellen, das Symbol eines Knotenpunktes für Konstruktionsfamilien von Knotenpunkten verwenden.

Zusammenfassend lassen sich so aus den vorausgegangenen Kapiteln folgende Aspekte zur Gestaltung und Programmierung von grafischen Benutzungsoberflächen resümieren, nach Priorität geordnet:

- Anfassers für direkte Manipulation, die keines erläuternden Textes bedürfen.
- Leitbilder als besonders effiziente Dialoge
- systemkonformes Erscheinungsbild aller Bildelemente, u.a. der Menü- und Symbolleisten, Dialogfenster, Schaltflächen etc.
- intuitiv verständliche Symbole für Menüs, Schaltflächen etc. anstelle von Beschriftungen
- intelligente Objekte, die sich mittels Maus direkt manipulieren lassen und dem ausgewählten Objekt dynamisch angepasste Dialogfenster
- einfache, intuitive Dialoge und Benutzerführung. z.B. durch eine einprägsame Vorgehensweise anstelle von mehreren
- Vermeidung von Modi
- keine Anhäufung von Symbolen und Symbolleisten
- keine ungewohnten Schaltflächen und -beschriftungen in Dialogfenstern

3.2 Entwurf einer effizienten Benutzungsoberfläche für Bausoftware

Bild 51 zeigt den in dieser Arbeit entwickelten Entwurf zur Lösung eines typischen Arbeitsbeispiels für ein intelligentes und intuitiv zu bedienendes CAD-System.

Wählt der Benutzer mit der Maus zwei Bauteile aus, zeigt das System sie durch eine Farbänderung als aktiviert an.

Wenn sich die Achsen der Stäbe schneiden, erkennt das CAD-System, dass die beiden Elemente durch eine CAD-Konstruktionsmethode miteinander verbunden werden müssen. Dies wird durch ein Symbol im Knotenpunkt angezeigt. Fährt der Benutzer mit der Maus über dieses Symbol, ändert sich der Mauscursor, wie bei Links in Internetseiten, in eine Hand, die dem Benutzer so signalisiert, dass er das Symbol anklicken kann.

In einem Dialogfenster, welches sich nach Klick auf das Knotensymbol öffnet, wählt der Benutzer aus der angebotenen Familie an Anschlüssen die gewünschte Bauart. Dabei werden als Familie nur die Konstruktionsmethoden angezeigt, die auf einen Knoten, in dem sich zwei Stäbe schneiden, angewendet werden können, also entsprechend der hier vorgefundenen Ausgangssituation. Die gewählte Methode wird als Leitbild angezeigt als Dokumentation, Hilfe und zur Direkteingabe.

Das Bestätigen des Dialogfensters über die Schaltfläche „OK“ führt den Anschluss durch die CAD-Konstruktionsmethode direkt auf die beiden Teile aus.

Das System erkennt also logisch den nächsten Schritt des Nutzers (er hat zwei Elemente ausgewählt, die sich in einem Knoten schneiden) und bietet ihm nur die Methoden zur Auswahl, die auf diese Situation anwendbar sind. Für den Benutzer ist ein effizientes, verständliches Vorgehen gesichert.

Die Bilder 52 und 53 zeigen ein weiteres typisches Arbeitsbeispiel und den entsprechenden Lösungsentwurf.

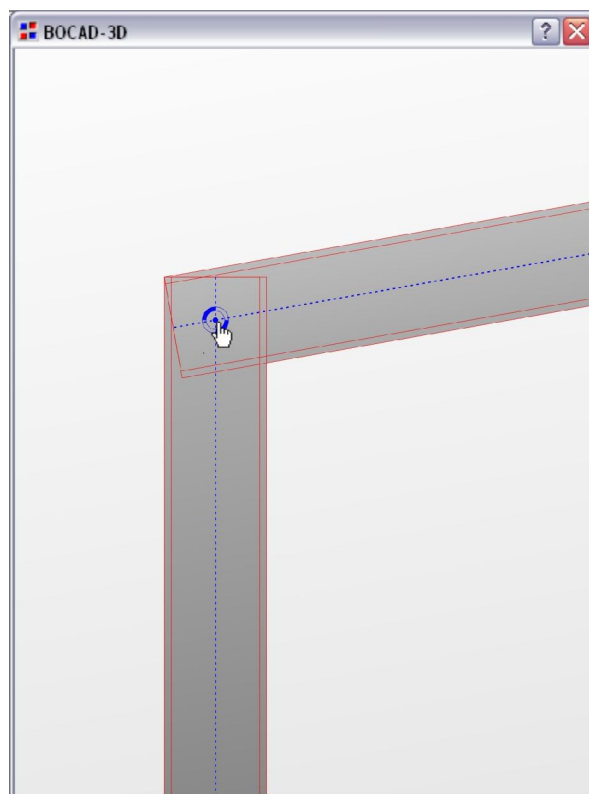


Abb. 51: Einfügen eines Anschlusses im Knotenpunkt zweier Stäbe ausgelöst durch weltweit bekanntes Handsymbol

Ein einzelnes ausgewähltes Element, z.B. ein Stab, wird durch Anfasser als aktiviert dargestellt. Mit der Maus kann der Benutzer, analog zu markierten Bildern in Word, das Objekt nun verschieben, kopieren oder in der Größe verändern (siehe Bild 52). Der Mauscursor ändert entsprechend der ausgeführten Aktion seine Form und signalisiert dem Benutzer so den aktuellen Zustand, wie bereits in [Chan02] und [Weck07] gefordert.

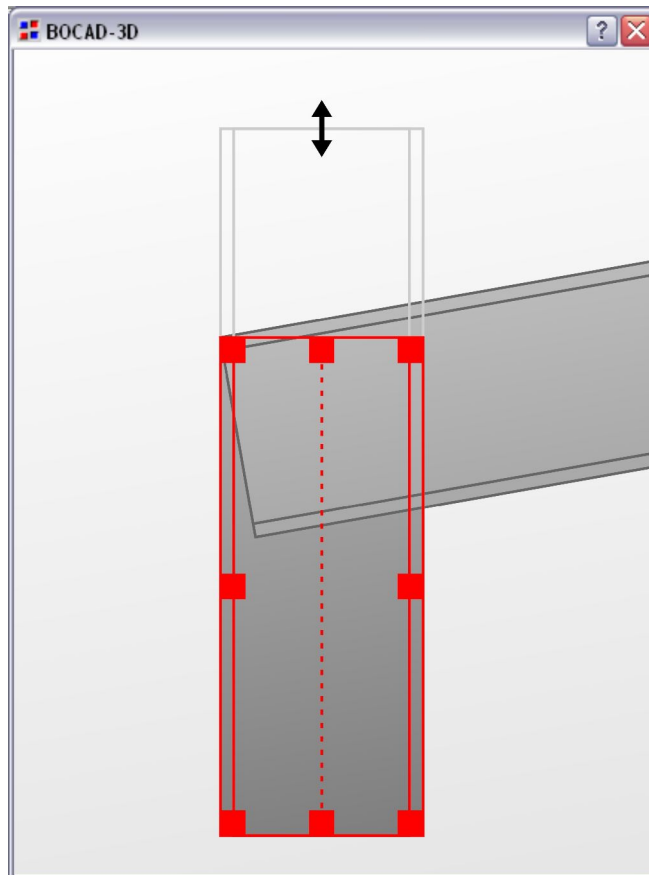


Abb. 52: Direkte Manipulation eines aktiven Objekts über Anfasser

Entsprechend internationaler Gewohnheit öffnet ein Doppelklick auf das aktive Element ein Dialogfenster, hier z.B. für die Eigenschaften eines Profilstabes. Es werden nur die Informationen angezeigt, die auf diesen Elementtyp anwendbar sind, so dass der Benutzer nicht durch überflüssige Angaben abgelenkt wird. Es ist identisch mit dem Dialogfeld, welches der Benutzer zum Platzieren des Elements benutzt. Er erkennt es daher wieder und erlernt den Umgang mit diesem entsprechend schnell.

Einen Entwurf für ein bis auf zwingende internationale Normbezeichnungen sprachfreies und intuitives Dialogfeld, welches nur die für das aktive Element wichtigen Daten enthält, zeigt Bild 53.

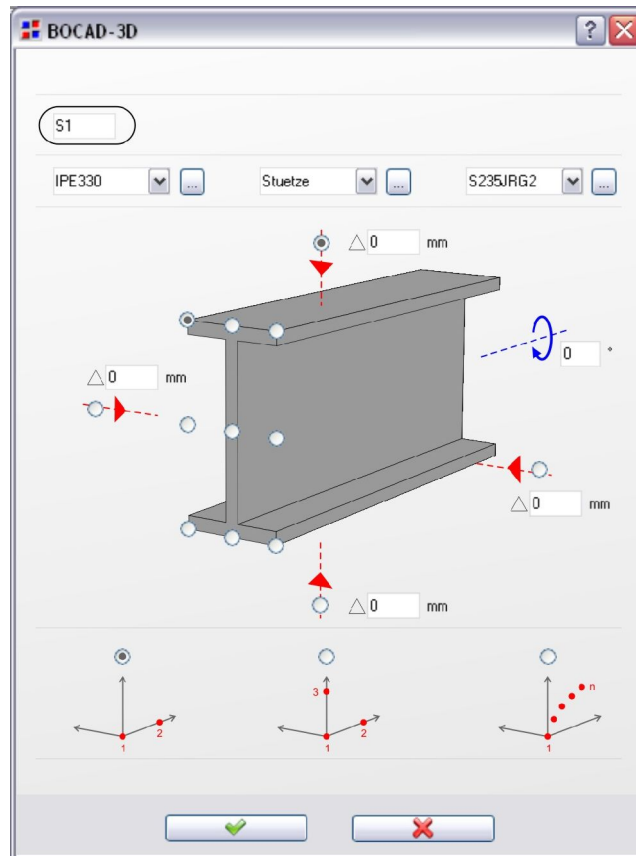


Abb. 53: Intuitives Dialogfenster zum Einfügen bzw. Bearbeiten eines Teils

In logischer Reihenfolge werden internationale Normbezeichnungen ausgewählt mit Profiltyp, Benennung und Material. Die Auswahl erfolgt über „Comboboxen“, die den Nutzer aus einer Liste ohne Möglichkeit von Schreibfehlern wählen lässt. Diese müssen für Fachleute des Bauwesens nicht durch Beschriftungen erklärt werden, denn ihr Inhalt ist selbsterklärend. So interpretiert der Benutzer „IPE330“ direkt als Profiltyp, ohne dass dies gesondert erläutert werden muss.

Eine räumliche Darstellung des Stabes im Dialogfenster zeigt die wählbaren Lage-Eigenschaften. Der Benutzer kann sie über Optionsfelder auswählen und über Eingabefelder zusätzliche Verschiebungen oder Drehungen eingeben.

Zuletzt wählt der Benutzer aus, über wie viele Punkte im Raum der Stab verlegt werden soll.

Lediglich zwei Befehlschaltflächen dienen dem Benutzer nun, die Änderungen zu bestätigen oder zu verwerfen. Auch diese kommen aufgrund der international geläufigen Symbole (grünes Häkchen für „so ausführen“ und rotes Kreuz für „abbrechen, nicht ausführen“) ohne weitere textliche Beschriftung aus.

Das beschriebene Dialogfeld kann vom Benutzer direkt, intuitiv und ohne Inanspruchnahme der Hilfefunktion („F1“-Taste) genutzt werden. Es ist wesentlich einfacher, schneller und fehlerfreier zu nutzen als die in BOCAD-3D oder Tekla Structures erörterten Dialogfelder und benötigt darüber hinaus bei internationalem Vertrieb der Software keinerlei Übersetzungsaufwand, vom Feld Benennung abgesehen. Der Inhalt des Feldes Benennung kann über eine Übersetzungstabelle automatisch in der korrekten Sprache eingefügt werden.

Die in Kapitel 5 entwickelten Software-Prototypen dieser Arbeit zeigen den programmtechnischen Ansatz auf, mit dem solche Aufgaben zu lösen sind.

Sie zeigen dabei zunächst die Programmierung von styleguide-konformen, intuitiv zu nutzenden Menüs und Symbolleisten sowie der entsprechenden Aktionen und Benutzerabfragen als klassischen Ansatz einer Benutzungsoberfläche. Ein weiterer Prototyp zeigt, wie die im klassischen Ansatz nicht zu vermeidenden Texte zu implementieren und im weiteren Verlauf der Softwareentwicklung in weitere Landessprachen und Schriften zu übertragen sind. Darüber hinaus zeigt dieser Prototyp, wie ein im Sinne dieser Arbeit innovativer neuer Ansatz für sprachfreie Menüs realisiert werden kann.

Zwei weitere Prototypen behandeln die Grafikprogrammierung anhand eines Grafikeditors, der das Platzieren von Grafikobjekten und die direkte sprachfreie Manipulation erlaubt. So zeigt speziell der dabei in dieser Arbeit entwickelte programmtechnische Ansatz, wie kleinere Softwarehäuser für Bausoftware dadurch wirtschaftlich internationale Software produzieren und vertreiben können.

4 Programmtechnischer Ansatz

4.1 Anforderungsprofil an Programmiersprachen

Für den programmtechnischen Ansatz dieser Arbeit sind die verfügbaren Programmiersprachen und Programmierwerkzeuge zu untersuchen, um gut begründet eine für das gesamte Anforderungsprofil weltweit verständlicher Bausoftware geeignete Programmiersprache auszuwählen.

Eine große Anzahl an Programmiersprachen steht zur Wahl, nur sehr wenige sind jedoch für diese Aufgabe zumindest grundsätzlich geeignet.

Pro Sprache wird daher im folgenden Abschnitt begründet, welche Spezifikationen des Anforderungsprofils zum Lösungsansatz dieser Arbeit erfüllt bzw. nicht erfüllt sind.

Das Anforderungsprofil ist entsprechend den Erkenntnissen der Kapitel 1 bis 3 wie folgt spezifiziert:

- 1) Zugänglichkeit grafischer Grundelemente der Benutzungsoberfläche, z.B. Anfasser
- 2) Verfügbarkeit einer vollständigen Bibliothek an grafischen Funktionen für die vorgestellten Elemente der Kapitel 1 bis 3
- 3) Verfügbarkeit eines Entwicklungswerkzeuges für Programmierung und Debugging zumindest auf dem Niveau von „Visual Basic for Applications“
- 4) Netzfähigkeit und Plattformunabhängigkeit

Auch moderne Programmiersprachen beruhen auf teils 50 Jahre alten Sprachen. Fortran, auf dem weite Teile des in dieser Arbeit betrachteten CAD-Hochleistungssystem BOCAD-3D basieren, geht zurück auf erste Entwicklungen im Jahr 1953.

Die weiterhin große Verbreitung dieser alten Programmiersprache in aktuellen Softwareprodukten sowie die noch stete Weiterentwicklung sind auf die umfangreichen Bibliotheken für wissenschaftliche und numerische Berechnungen zurückzuführen, die auch heute noch unentbehrlich sind [WIKI-Fortran].

Aufgrund fehlender Grafikbibliotheken ist Fortran allerdings nicht für die Gestaltung einer grafischen Benutzungsoberfläche im Sinne dieser Arbeit geeignet.

VisualBasic und speziell VBA besitzen einige objektorientierte Eigenschaften. Sie verfügen jedoch nicht über alle Kriterien einer objektorientierten Programmiersprache. Die „Einfachheit von BASIC kommt all jenen Menschen entgegen, die den Computer zwar

verwenden, aber nicht das Programmieren lernen möchten“ [Zuse99]. So kam es, dass BASIC eine bedeutende Rolle im Informatikunterricht einnahm und heute mit Visual Basic for Applications zeitgemäß gelehrt wird.

```
Sub Hallo
    MsgBox "Hallo Welt!"
end sub 'Hallo
```

Abb. 54: Hallo-Welt-Programm in VisualBasic.

Dieses Programm erzeugt für Dialogmeldungen bereits eine grafische Ausgabe (siehe Bild 55):



Abb. 55: Grafische Ausgabe "Halo Welt" von VBA

In den frühen 1970er Jahren wurde die Programmiersprache C von Ken Thompson und Dennis Ritchie entwickelt, damals für das neu entwickelte Betriebssystem UNIX [WIKI-C]. C steht heute auf fast allen Computersystemen zur Verfügung, da viele der Betriebssystemkerne auf C basieren. So ist C auch die Basis für die Programmierschnittstellen (API) von Windows. Die genormte Standard-C-Bibliothek steht so auf jedem C-System zur Verfügung. Diese Normung wurde 1989 erstmals durchgeführt und im Jahr 1999 überarbeitet.

```
# include <stdio.h>

int main(void)
{
    printf("Hallo Welt!\n");
    return 0;
}
```

Abb. 56: Hallo-Welt-Programm in C, Quelle: [WIKI-Hallo]

Die Weiterentwicklung C++ wurde ab 1979 von Bjarne Stroustrup bei AT&T entwickelt. Primäres Ziel war es damals, die verbreitete Programmiersprache C um ein Klassenkonzept zu erweitern und Erfahrungen aus der Promotionsarbeit von Stroustrup einfließen zu lassen [WIKI-C++].

```
#include <iostream>

int main()
{
    std::cout <<"Hallo Welt!" << std::endl;
}
```

Abb. 57: Hallo-Welt-Programm in C++, Quelle: [WIKI-Hallo]

Im Jahr 1998 wurde die endgültige Fassung der Sprache von der ISO genormt und somit weltweit allgemein gültig.

Gängige Windowsprogramme werden vielfach in C++ geschrieben unter Nutzung der Microsoft Foundation Classes (MFC) zur Gestaltung der Benutzungsoberfläche. Aufgrund dieser MFC sind diese Programme allerdings nicht plattformübergreifend nutzbar.

Microsoft entwickelte ab dem Jahr 2000 die Programmiersprache C# (sprich: C sharp), dass trotz ähnlicher Namensgebung nicht kompatibel zu C++ ist. C# ist eine rein objektorientierte Sprache und greift Konzepte aus Java, Delphi, C++ und auch VisualBasic auf. Seit 2003 ist C# von der ISO standardisiert [WWW-C#].

```
class HalloWeltApp
{
    public static void Main()
    {
        System.Console.WriteLine("Hallo Welt!");
    }
}
```

Abb. 58: Hallo-Welt-Programm mit C#, Quelle: [WIKI-Hallo]

C# ist im Gegensatz zu C und C++ deutlich vereinfacht. So wurden Zeiger abgeschafft, die nur noch in sog. „unsafe code“ explizit freigegeben werden können [WIKI-C#].

Aufgrund der Integration in das .NET-Framework von Microsoft ist C# nicht plattformunabhängig. Plattformunabhängigkeit ergibt sich aber durch das Gegenstück zum .NET-

Framework von Microsoft, das freie Projekt MONO. Dieses wird mitlaufend weiterentwickelt und dauerhaft für Linux-Systeme angeboten.

Ebenfalls im Rahmen des .NET-Frameworks entwickelte Microsoft im Jahre 2002 Visual Basic weiter zu VisualBasic.NET. Es handelt sich nun um eine vollständig objektorientierte Sprache.

VB.NET erstellt, ebenso wie C# und Java, einen Zwischencode, die sog. MSIL (Microsoft Intermediate Language). Dieser Zwischencode wird dann von einer virtuellen Maschine, der Common Language Runtime, ausgeführt.

VisualBasic.NET eignet sich sowohl für die Erstellung von Windows-Applikationen als auch für Webanwendungen, für die dann ASP.NET verwendet wird [WIKI-VB.NET]. Auch hier ist die Plattformunabhängigkeit durch MONO gegeben.

```
Class HalloWelt
  Shared Sub Main()
    Console.WriteLine("Hallo Welt")
  End Sub
End Class
```

Abb. 59: Hallo-Welt-Programm mit VB.NET, Quelle: [WIKI-Hallo]

Die Programmiersprache Java entstand bereits Anfang 1991 im Auftrag von Sun. Sie ist objektorientiert und plattformunabhängig. Java-Programme benötigen daher die sog. „Java Virtual Machine“, die für zahlreiche Betriebssysteme verfügbar ist.

Ursprüngliches Ziel war nicht die Entwicklung einer weiteren Programmiersprache, sondern die Entwicklung einer vollständigen Betriebssystemumgebung inkl. virtueller CPU, um – einer Legende nach – selbst Kaffeemaschinen steuern zu können [WIKI-Java].

Der Erfolg von Java stellte sich erst mit der Integration in den Browser Netscape ein.

Im Rahmen dieser Arbeit ist relevant, dass Java vergleichsweise langsame, nicht akzeptable Reaktionszeiten bewirkt. Dies liegt an der Java Virtual Machine, die als Zwischenschicht das Programm erst zur Laufzeit kompiliert. Auch mit Java 2.0 sind die Programme lediglich vorkompiliert und erreichen auch in dieser Version nicht die Geschwindigkeit komplett kompilierter Programme.

Anwendungen in Java können ohne Quelltextanpassung auf verschiedenen Plattformen ausgeführt werden, sofern für diese eine Java Virtual Machine verfügbar und installiert ist.


```
public class Hallo {  
    public static void main(String[] args) {  
        System.out.println("Hallo Welt!");  
    }  
}
```

Abb. 60: Hallo-Welt-Programm in Java, Quelle: [WIKI-Hallo]

Trotz Namensähnlichkeit ist JavaScript eine gänzlich andere Programmiersprache als Java, die nicht mit Java konkurriert. JavaScript ist eine objektorientierte Skriptsprache, die von Netscape entwickelt wurde um HTML-Seiten dynamisch gestalten zu können. Der Name wurde lediglich aus Marketinggründen gewählt [WIKI-JS].

Keine Programmiersprache, sondern eine Klassenbibliothek für die plattformübergreifende Programmierung grafischer Benutzungsoberflächen unter C++ ist Qt. Speziell unter Linux für die Programmierung der sog. „K Desktop Environments“ (dem Gegenstück zur Windows-Oberfläche zur Systemverwaltung) verwendet, ist Qt auch in klassischen Windowsprogrammen schon eingesetzt worden und bewährt sich dort.

Neben der Plattformunabhängigkeit bei der Gestaltung grafischer Benutzungsoberflächen bietet Qt zudem umfangreiche Funktionen zur Internationalisierung von Software an.

Qt wurde 1991 von der norwegischen Firma Quasar Technologies (heute Trolltech) entwickelt [WIKI-Qt]. Während die Linux-Version von Qt stets unter der GPL (*General Public License*, eine Lizenzierung für freie Software) veröffentlicht wurde, waren die Windows- und Mac-Version nur als kommerzielle Lizenz erhältlich. Für freie Software, die für die beiden Betriebssysteme entsteht, bietet Trolltech seit 2005 ebenfalls eine Lizenzierung unter der GPL an.

Das Lehr- und Forschungsgebiet Bauinformatik hat zu Forschungszwecken eine derartige Lizenz angeschafft und experimentell die Anwendung erprobt.

Aufgrund seiner mustergültigen Implementation in die Entwicklungsumgebung Microsoft VisualStudio und der Möglichkeit, plattformübergreifende Anwendungen zu entwickeln werden auch einige der im fünften Kapitel vorgestellten Softwareprototypen in C++ und Qt programmiert.

Während der Entwicklung dieser Prototypen musste allerdings festgestellt werden, dass mit Qt der wesentliche Kern dieser Arbeit, die Programmierung von Anfassern für die direkte, interaktive Manipulation von Objekten, derzeit nicht erreicht werden kann. Dies wurde von Trolltech schriftlich bestätigt. Erst Anfang 2007 konnte diese Problematik mit dem Erscheinen von Windows Vista und der damit neu geschaffenen Windows Presentation Foundation (WPF) und der neuesten Version des .NET-Framework mustergültig gelöst werden. Auch diesen Lösungsweg zeigt das fünfte Kapitel auf.

4.2 Lösungsweg

Die Grafik-Programmierung, also die Entwicklung von Programmen, die unter einer grafischen Benutzungsoberfläche ausgeführt werden, unterscheidet sich ganz wesentlich von der Programmierung von sog. Konsolen-Programme (auch textbasierte Programme genannt), wie sie mit den „Hallo Welt“-Programmen im vorherigen Kapitel vorgestellt wurden.

Während diese prozedural gesteuert sind, d.h. sie reagieren auf Befehle, deren Abfolge in einer zeitlichen oder logischen Reihenfolge durch das Programm vorgegeben wird, sind Grafik-Programme ereignisgesteuert (auch *Message*-gesteuert genannt). Hierbei empfängt das Programm Nachrichten vom Betriebssystem, die es darüber informieren, welches Ereignis der Benutzer ausgelöst hat. Aufgabe der Anwendung ist es nun, auf diese Nachricht (*Message*) entsprechend zu reagieren [Rieg05].

So führt z.B. ein Klick mit der linken Maustaste in ein Programmfenster dazu, dass das Betriebssystem feststellt, dass

- ein Mausklick erfolgt ist
- ein einfacher Mausklick mit der linken Maustaste erfolgt ist
- in das Programmfenster geklickt wurde

Bei der Übermittlung der Nachricht wird zudem die Mausposition übermittelt, so dass das Programm ermitteln kann, ob und welche Schaltfläche oder Objekt angeklickt wurde.

Neben diesen Nachrichten sendet das Betriebssystem auch stets Informationen über die aktuelle Position der Maus nachdem diese bewegt wurde oder über etwaige Tastendrücken.

cke an die Nachrichten-Warteschlange (*Message Queue*). In diesen Puffer laufen so alle Nachrichten ein und werden nacheinander abgearbeitet, d.h. an das die Nachricht betreffende Programmfenster gesendet.

Durch die Pufferung gehen keine Eingaben des Benutzers verloren. Ist z.B. eine Textanwendung durch eine Rechenoperation momentan so beschäftigt, dass es keine Nachrichten annehmen kann, werden alle Tastendrücke gepuffert und, nachdem das Programm seine Rechenoperation beendet hat, an dieses gesendet.

All dies bedeutet bei der Grafik-Programmierung ein Umdenken und eine andere Herangehensweise. Wissenschaftliche Literatur, die sich mit diesem Thema auseinandersetzt, ist bisher spärlich und anspruchsvoll, also zumindest für die Lehre im Grundstudium wenig geeignet (z.B. [Balz00], [Rieß06], [Booc94])



Abb. 61: Exemplarische, internationale Benutzungsoberfläche aus der Lehre am Lehr- und Forschungsgebiet Bauinformatik

Um das entsprechende Grundverständnis für diesen Mechanismus der ereignisgesteuerten Programmierung als elementare Grundlage heutiger, interaktiv grafischer Benutzungsoberflächen zu wecken und die Konsequenzen für praxistaugliche Bausoftware zu verdeutlichen, erarbeiten die Studierenden des Bauingenieurwesens und der Sicherheitstechnik daher zu Beginn des Studiums exemplarische Benutzungsoberflächen zu fachbezogenen Anwendungsprogrammen. Dazu wird in der Lehre die in allen Microsoft

Office-Anwendungen verfügbare Programmiersprache VisualBasic for Applications (VBA) eingesetzt. Bild 61 zeigt die Benutzungsoberfläche eines derartigen Beispiels aus der Lehre.

Der einfachste Fall, der in der Lehre das Prinzip von Ereignisprogrammen prägnant darstellt, ist der Abbruch des Programms auf Wunsch des Benutzers. Dazu klickt der Benutzer auf die Befehlsschaltfläche „Abbrechen“, hier beschriftet durch ein rotes X. Damit löst er das Ereignisprogramm `Abbrechen_Click()` aus. Dies enthält lediglich die Anweisung „End“ und beendet somit den Programmlauf (siehe Bild 62).

```
Private Sub Abbrechen_Click()  
    End      'Bricht das Programm ab  
End Sub
```

Abb. 62: Prägnantes Ereignisprogramm aus der Lehre

Die Verwaltung der Nachrichten, die das Betriebssystem permanent sendet, übernimmt dabei Microsoft Office. So erhalten die Studierenden einen elementaren Einstieg in die Programmierung von grafischen, ereignisgesteuerten Anwendungen.

Bei der Programmierung von komplexen, eigenständigen Programmen (VBA-Programme sind nur innerhalb von Microsoft Office lauffähig), muss das Programm über eine Schleife die Nachrichten-Warteschlange abfragen und mit entsprechenden Funktionen verarbeiten.

Besonders hervorzuheben für das Ziel dieser Arbeit ist der umgekehrte Weg, dass nämlich das Anwendungsprogramm als Reaktion auf eine Nachricht den Mauszeiger auf dem Bildschirm ändern kann, z.B. wenn der Benutzer ein Objekt ausgewählt hat (linke Maustaste gedrückt und gehalten, Objekt dabei angeklickt) und dann bewegt wird (Mausbewegung).

Bereits an diesen Beispielen wird deutlich, dass die Grafik-Programmierung für Benutzungsoberflächen komplex ist. Auf jede mögliche Benutzerinteraktion muss eine entsprechende Abfrage und Reaktion programmiert werden, wobei selbst unvorhersehbare Aktionen des Benutzers nicht zu falscher Reihenfolge bei der Abarbeitung zeitkritischer

Ereignisse oder zu Deadlocks, gegenseitiger Blockade von Ereignisprogrammen, führen dürfen.

Einen entsprechenden programmtechnischen Ansatz für eine praktische Anwendung zeigen die im folgenden Kapitel vorgestellten Software-Prototypen.

4.3 Grundlagen der objektorientierten Programmierung

Zur besseren Verständlichkeit der objektorientierten Programmierung wird zunächst ihr Grundkonzept vorgestellt.

Nach [Oest06] ist jeder Gegenstand in der realen Welt ein Objekt im Sinne der Objektorientierung. So ist ein Telefon ein Objekt, ebenso wie ein Fahrrad oder ein Auto. Jedes dieser Objekte setzt sich aus weiteren Objekten zusammen, z.B. Schrauben, Rädern usw.

In der Softwareentwicklung werden diese realen Objekte modellhaft dargestellt, d.h. auf die für die Aufgabe bedeutsamen Eigenschaften reduziert.

So werden nicht alle gleichartigen Objekte individuell modelliert, sondern es wird lediglich ein Bauplan der Objektfamilie erstellt. Dieser wird in der Objektorientierung Klasse genannt. Mit Hilfe dieser Klasse werden dann die konkreten Objekte als einzelne Familienmitglieder erzeugt. Diese werden als Exemplar oder Instanz bezeichnet. So wird gesagt „Das gelbe Auto ist eine Instanz der Klasse Auto“.

Objekte, also instanzierte Exemplare einer Klasse, kapseln ihre Informationen (Daten) als Attribute und ihr Verhalten als Methoden. Ein Objekt gehört genau einer Klasse an, es kann seine Klassenzugehörigkeit nicht ändern.

Klassen sind also eine allgemeine Beschreibung für alle Objekte, die dieser Klasse angehören. In dieser sind sowohl die den Objekten gemeinsamen Attribute und Methoden definiert als auch eine Beschreibung zur Erzeugung neuer Objekte dieser Klasse. Die Beschreibungen zur Erzeugung neuer Objekte werden Konstruktor genannt und aufgerufen, wenn ein neues Objekt erstellt werden soll. Der Destruktor löscht entsprechend ein Objekt.

Nach [Bann99] spezifiziert eine Klasse demnach die gemeinsamen Eigenschaften und Verhalten der von ihr erzeugten Objekte. Die Klassenbeschreibung umfasst dabei das Interface (die öffentliche Schnittstelle, mit der das Objekt mit anderen Objekten kommunizieren kann) und die Implementierung der Methoden und Klasseneigenschaften. Die Schnittstelle besteht dabei aus den Deklarationen aller Operationen, die auf Instanzen der Klasse angewendet werden können. Sie teilt sich in drei Teile. Im *public*-Bereich werden Methoden deklariert, auf die alle Objekte zugreifen dürfen. Unter *protected* finden sich Methoden, auf die nur die Klasse selbst, ihre Unterklassen und *Friends* (als „befreundet“ definierte Klassen) zugreifen dürfen. Auf Deklarationen die unter *private* getroffen werden, kann nur die Klasse selbst und ihre als „befreundet“ deklarierten Klassen (*Friends*) zugreifen.

Klassen können darüber hinaus noch Klassenvariablen und Klassenfunktionen besitzen, die nicht mit einem individuellen Objekt verbunden sind. Diese sind nur innerhalb der Klasse aufzurufen und nutzbar. Im Gegensatz zu den Membervariablen (eine weitere Bezeichnung für Attribute) stehen sie bereits zur Verfügung, wenn noch kein Objekt erzeugt wurde.

Attribute bezeichnen die Eigenschaften von Objekten, in denen die Daten gespeichert werden. Attribute können wiederum komplexe Datenstrukturen oder Objekte sein. Attribute können von außen nur über die Methoden der Schnittstelle gelesen oder geändert werden.

Methoden sind Funktionen, mit denen ein Objekt (bzw. dessen Attribute) verändert werden werden. Dabei wird Methode vielfach synonym mit Funktion verwendet. Zusammen mit Attributen gehören sie zu den Objekteigenschaften.

Methoden werden über einen Funktionsaufruf ausgeführt der als Senden einer Nachricht bzw. als Botschaft bezeichnet wird. Methoden sind an ihre Klassen gebunden und lassen sich, anders als bei klassischen Funktionen oder Unterprogrammen, nur im Zusammenhang mit einem Objekt aufrufen.

Bild 63 zeigt als triviales Beispiel die Klassendefinition der Klasse *Auto*. Das Exemplar „gelbes Auto“ hat die Eigenschaft „farbe = gelb“. Auf dieses Attribut kann nicht direkt zugegriffen werden, da es in der Klasse gekapselt ist. Dies ist nur über die Methode „setzeFarbe()“ möglich, die die Eigenschaft des Attributs ändern kann.

Klassenname	Auto
Attribute	farbe reifentyp anzahl_tueren
Methoden	setzeFarbe() wechsleReifen()

Abb. 63: Klassendefinition der Klasse Auto

Ein wesentliches Leistungsmerkmal der objektorientierten Programmierung ist die Vererbung. Mittels Vererbung können speziellere Unterklassen gebildet werden oder allgemeine Eigenschaften in eine Oberklasse verlagert werden. Vererbung ist somit ein hervorragendes Strukturwerkzeug.

Von der Klasse „Auto“ sind so weitere Unterklassen „Limousine“, „Kombi“ und „Cabrio“ abzuleiten. Unterklassen besitzen neben den gemeinsamen Eigenschaften ihrer Basis-klasse zusätzlich ihre speziellen Eigenschaften.

Nach [Bann99] lässt sich ein objektorientiertes Programm somit als ein System kommunizierender Objekte zusammenfassen, die sich gegenseitig benachrichtigen und Methoden aufrufen, um ihre Daten objektkonform gesichert im Inneren zu verarbeiten.

Eben dieses Prinzip der objektorientierten Programmierung erlernen die Studierenden des Bauingenieurwesens und der Sicherheitstechnik an der Bergischen Universität Wuppertal anhand beispielhafter Programme in VBA bereits im ersten Semester. Dabei übernimmt die Programmierumgebung von VBA große Teile des programmtechnischen Unterbaus. Der Benutzer muss lediglich Ereignisprogramme, Dialogfelder und Algorithmen programmieren. Dieser effiziente Einstieg in die Programmierung moderner Programmsysteme, die mittels Maus vom Benutzer gesteuert werden, hat sich in der Bauinformatik fast an allen Universitäten durchgesetzt.

5 Experimenteller Nachweis und Praxiserfahrungen

5.1 Rahmenbedingungen und eingesetzte Werkzeuge

Anhand der Erkenntnisse aus Kapitel 2 bis 4 werden Software-Prototypen konzipiert, mit denen die Eignung des programmtechnischen Ansatzes für effiziente und weltweit verständliche Benutzungsoberflächen experimentell nachgewiesen wird.

Für die ersten, einführenden Prototypen wird die plattformübergreifende GUI-Bibliothek Qt der Firma Trolltech verwendet zusammen mit der Programmiersprache C++. Als Entwicklungsumgebung dient Microsofts VisualStudio .NET 2003 mit der Qt VisualStudio Implementierung und Qt in Version 4.1.0.

Auch diese sehr modernen Werkzeuge genügten den Anforderungen des Arbeitsthemas letztlich nicht vollständig. Es fehlte die Unterstützung der direkten Manipulation durch Anfasser.

Deshalb wurde ab Januar 2007 das Microsoft .NET-Framework in Version 3.0 mit der Programmiersprache C# sowie die mit dem Erscheinen von Windows Vista Anfang 2007 neu geschaffene Programmierschnittstelle Windows Presentation Foundation (WPF) mit der neuen Sprache XAML verwendet. Als Entwicklungsumgebung dient Microsofts VisualStudio C# 2005 Express Edition, welche frei verfügbar ist, sowie das noch im Beta-Stadium befindliche Microsoft Expression Blend als Gestaltungssoftware für die Oberflächenelemente.

Die neueste Generation VisualStudio 9.0 wurde im Rahmen dieser Arbeit ebenfalls evaluiert. Es ist (Stand Februar 2007) als Community Technology Preview (CTP) verfügbar, und wird später die Funktionen des in dieser Arbeit eingesetzten VisualStudio C# 2005 Express Edition und Expression Blend in einer Software kombinieren.

Auch den Entwicklern der GUI-Bibliothek Qt der Firma Trolltech ist durch den Briefwechsel mit dem Verfasser offenkundig geworden, dass sie die Anforderungen effizienter und verständlicher Benutzungsoberflächen der Zukunft in ganz wesentlichen Punkten bisher nicht erkannt hatten. Sie wollen die entsprechenden Entwicklungen nun so rasch wie möglich nachholen, können jedoch keinen kurzfristigen Liefertermin zusagen, da die Entwicklung dieser zukunftsweisenden Werkzeuge in der Tat sehr komplex und von hohem Schwierigkeitsgrad ist.

Gelingt Trolltech diese durch universitäre Forschung angestoßene Zukunftsentwicklung, sind die Anforderungen des Arbeitsthemas plattformübergreifend zu lösen.

Ansonsten ist seit Anfang 2007 zumindest eine Lösung in der Windows-Welt funktions-tüchtig machbar, wie im abschließenden Prototyp für direkte Manipulation mit Anfassern nachgewiesen.

5.2 Software-Prototyp I: Menü- und Symbolleisten am Beispiel eines Texteditors

In diesem ersten Beispiel wird zunächst der programmtechnische Ansatz einer symbol-unterstützten klassischen Benutzungsoberfläche (siehe Bild 64) vorgestellt. So ist ein Texteditor für die letztlich verbleibenden, unvermeidlichen Texte in ansonsten textfreien Benutzungsoberflächen notwendig, denn auch für die verbleibenden Texte, wie z.B. Produktbezeichnungen für Zukaufteile, verwendete Normen etc., muss schließlich eine effiziente Lösung entwickelt und angeboten werden.

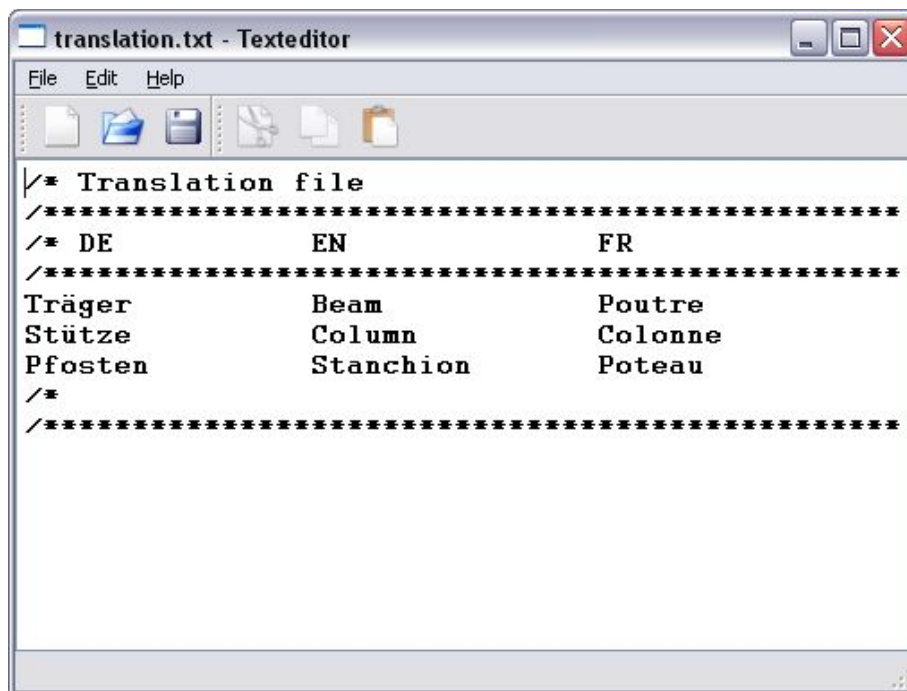


Abb. 64: Software-Prototyp "Texteditor"

Dieser Softwareprototyp zeigt den programmtechnischen Ansatz auf, klassische Menüs, Symbolleisten und Kontextmenüs möglichst nah an den Zielen dieser Arbeit zu program-

mieren. Die Verwendung von identischen Icons sowohl in den Menüs als auch in den Symbolleisten macht die Bedienung für den Benutzer intuitiv und entspricht so den in Kapitel 2 hergeleiteten Spezifikationen für klassische Menüs.

Qt vereinfacht die Programmierung von Menüs und Symbolleisten durch das „Aktionen“-Konzept. Als Aktion sind die zusammengehörigen Elemente Betextung, Symbol, Quickinfo, Zugriffstaste und das Tastaturkürzel definiert, siehe Bild 65.

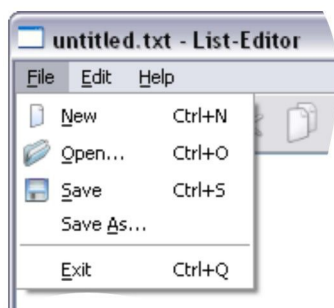


Abb. 65: Menüeinträge im Software-Prototyp I

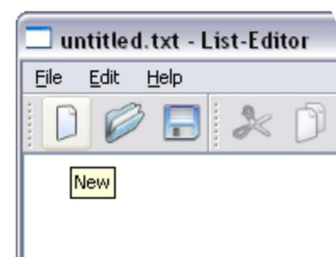


Abb. 66: Symbole im Software-Prototyp I

Nach Definition einer Qt-Aktion wird diese anschließend den betroffenen Objekten (Menüs und Symbolleisten) hinzugefügt. Dadurch wird dem Benutzer je nach Kontext die entsprechende Information angezeigt: Die Betextung, links davon das Icon und rechts daneben das Tastaturkürzel. Die Zugriffstaste wird mit einem Unterstrich hervorgehoben (siehe Abb. 65).

In der Symbolleiste (siehe Abb. 66) werden Aktionen nur als Icon dargestellt. Verharrt die Maus einige Zeit über dem Icon wird die Betextung als Quickinfo angezeigt.

Die Programmierung des Prototyps erfolgt in der geeigneten Sprache C++.

Ein C++-Programm besteht aus einer oder mehreren Kompilierungseinheiten mit der Dateiendung `.cpp`. Diese werden durch Kompilieren zu Objektdateien, die dann vom Linker zu einem ausführbaren Programm verkettet werden. Die Kompilierungseinheiten sind frei wählbar, z.B. ob sie jeweils nur eine Klasse enthalten oder mehrere. Genau

eine Kompilierungseinheit muss die Funktion `main()` enthalten, die als Eintrittspunkt in das Programm dient.

Alle C++-Programme besitzen Headerdateien mit der Dateiendung `.h`. Diese können in die Kompilierungseinheit mit `include` eingefügt werden und stehen dann in diesen zur Verfügung. Headerdateien werden beim Kompilieren nicht zu Objektdateien gewandelt.

Das Programmbeispiel beginnt mit der Funktion `main()` als Eintrittspunkt in der Datei `main.cpp`, siehe Bild 67. Zunächst werden zwei Headerdateien inkludiert. `QApplication` steht in spitzen Klammern. Dies bedeutet, dass es aus der Standard Qt-Bibliothek einzufügen ist. Es folgt das Inkludieren von `mainwindow.h`, der noch zu definierenden Klasse.

```
//Inhalt der Datei main.cpp
//inkludieren von Headerdateien
#include <QApplication>
#include "mainwindow.h"

//Typischer Aufruf eines Hauptprogramms in C++
//Funktion main übernimmt einen Wert vom Typ int auf argc und einen Wert
//vom Typ char auf dem Array argv[]
int main(int argc, char *argv[])
{
//Qt-Ressource application wird initiiert
    Q_INIT_RESOURCE(application);
//Der QApplication app werden die Werte von argc und argv übergeben
    QApplication app(argc, argv);
//Instanzieren des Objekts mainWin aus der Qt-Klasse MainWindow
    MainWindow mainWin;
//Anzeigen des Objektes mainWin
    mainWin.show();
//Funktion main gibt app.exec() zurück, d.h. führt Application aus
    return app.exec();
}
```

Abb. 67: Hauptprogramm in `main.cpp`

Nach [Blanc01] ist das Hauptprogramm nach Bild 67 typisch für C++-Programme. Die Funktion `main` hat zwei Parameter, eine `int`-Variable und ein Array `*argv[]`. Die Namen dieser Parameter von `main`, `argc` (Argument count) und `argv` (Argument Value), sind in C++ vorgegeben. Der Name des Programms (hier `main`) ist in Feld `argv[0]` verfügbar, weitere Befehlszeilenargumente unter `argv[1]`, `argv[2]`, ..., `argv[argc-1]`. Da dieses Programm nicht über die Befehlszeile ausgeführt wird, sind hier die Parameter nicht relevant, aber formal erforderlich.

Im Ausführungsteil der Funktion `main()` wird zunächst die Initialisierung der Qt-Ressourcen vorgenommen durch den Aufruf der Qt-Funktion `Q_INIT_RESOURCE(application)`. Es folgt die Erstellung eines `QApplication`-Objektes `app` zur Verwaltung der Anwendungsressourcen sowie des Hauptfensters namens `mainWin`. Durch `mainWin.show()` wird dieses Objekt auf dem Bildschirm angezeigt, analog zur Funktion `show` in VBA.

```
//Ableiten von MainWindow von der Qt-Basisklasse QMainWindow
class MainWindow : public QMainWindow
{
//Makro Q_OBJECT stellt Signale und Slots für Qt bereit
    Q_OBJECT

//Definition des Konstruktors im public-Bereich
public:
    MainWindow();

//Methodendeklaration im Bereich protected
protected:
    void closeEvent(QCloseEvent *event);

//In private slots werden Qt-Slots als Methoden definiert.
private slots:
    void newFile();
    void open();
    (...)

//Attribute und Methoden im Bereich private
private:
    void createActions();
    void createMenus();
    void writeSettings();
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    (...)
//Definition der Attribute, z.B.
//der Zeiger textEdit vom Typ QTextEdit
    QTextEdit *textEdit;
    QString curFile;

    QMenu *fileMenu;
    QMenu *editMenu;
    QMenu *helpMenu;
    (...)
};
```

Abb. 68: Individuelle Definition der Klasse `MainWindow`

Die Klassendefinition der hier genutzten Klasse `MainWindow` erfolgt wie in C++ gewöhnlich in einer Headerdatei, in diesem Fall `mainwindow.h`, siehe Bild 68. Sie wird in

die Abschnitte `public`, `private` und `protected` unterteilt, sowie in `private slots` für die Definition der Slots für die später noch erläuterten Qt-Signale.

In `mainwindow.h` wird die Klasse `MainWindow` durch Ableitung aus der Qt-Standardklasse `QMainWindow` definiert. Sie erbt damit alle Funktionen dieser Klasse. Dies sind u.a. Funktionen zur Programmierung von Menü-, Symbol oder Statusleisten. Aus diesem Grunde eignet sich dieses Beispiel musterhaft für den Einstieg in die objektorientierte Anwendungsentwicklung von Benutzungsoberflächen klassischer Art.

`MainWindow` besitzt so alle `public`-Eigenschaften und Methoden dieser Standardklasse und wird durch weitere Attribute und Methoden erweitert, wie Bild 68 zeigt.

In der Klassendefinition werden dabei nur die Funktionsprototypen definiert, d.h. der Name der Funktion und der erwartete Rückgabewert der Funktion. Ebenso verhält es sich für die Attribute. Auch für diese wird nur der Name und der Speichertyp festgelegt.

Als Qt-Spezifikation besitzt die Klasse zudem den Bereich `private slots`. In diesem werden alle Slots, über die die Klasse verfügen soll, definiert. Diese werden später in der Implementierung der Klasse von entsprechend ausgelösten Signalen aufgerufen.

So werden z.B. die Attribute `fileMenu` und `editMenu` als Zeiger auf einen Datentyp `QMenu` definiert, um in der Implementierung der Klasse auf diesen Attributen Menüeinträge speichern zu können. Die Funktionen `createActions()` oder `writeSettings()` werden als `void`, also ohne Rückgabewert, definiert. Durch Definition im `private`-Bereich sind diese Attribute und Methoden von außen nicht sichtbar und somit vor unbefugtem Zugriff geschützt.

Die Implementierung der Klasse `MainWindow` erfolgt in Datei `mainwindow.cpp`. Zu Beginn müssen die Klassendefinitionen der QtGui- und QFont-Bibliotheken eingebunden werden, ebenso wie die zuvor erläuterte Klassendefinition von `MainWindow`, siehe Abb. 69.

```
#include <QtGui>
#include <QFont>

#include "mainwindow.h"
```

Abb. 69: Implementierung der Klasse `MainWindow` durch Inkludieren der notwendigen Headerdateien

Bild 70 zeigt den Konstruktor der Klasse, eine Funktion zur Erzeugung neuer Objekte einer Klasse, wie bereits in Kapitel 4.3 erläutert.

Um ein neues Objekt der Klasse MainWindow zu erzeugen (instanzieren), führt der Konstruktor folgende Arbeitsschritte durch:

- Er erzeugt ein Objekt `textEdit` als zentrales Element (Widget, nach Qt-Definition ein Bedienelement einer grafischen Benutzungsoberfläche) der Anwendung
- er ruft die Funktionen `createActions()`, `createMenus()`, `createToolBars()` und `createStatusBar()` auf
- er ruft die Methode `readSettings()` auf
- er verbindet ein Signal mit einem entsprechende Slot und
- er ruft die Funktion `setCurrentFile("")` auf.

```
MainWindow::MainWindow()
{
    //ein neues Objekt textEdit wird aus Klasse QTextEdit erstellt (Aufruf
    //des Konstruktors
    textEdit = new QTextEdit;
    //textEdit wird zum zentralen, das Anwendungsfenster füllenden Element
    setCentralWidget(textEdit);
    //Schriftart wird gesetzt
    QFont f("Courier", 12, QFont::Bold);
    setFont(f);

    //Aktionen, Menüs, Symbolleisten und Statuszeile werden erstellt
    //Dazu werden die entsprechenden Funktionen aufgerufen
    createActions();
    createMenus();
    createToolBars();
    createStatusBar();

    //Benutzereinstellungen werden gelesen, d.h. Funktion readSettings()
    //wird aufgerufen
    readSettings();

    //Signal-Slot Verbindung zwischen textEdit, wenn dieses geändert wurde,
    //und Funktion documentWasModified
    connect(textEdit->document(), SIGNAL(contentsChanged()),
           this, SLOT(documentWasModified()));

    //ein leeres, unbenanntes Dokument wird als aktuell gesetzt
    setCurrentFile("");
}
```

Abb. 70: Konstruktor der Klasse MainWindow

`TextEdit`, als zentrales Element der Anwendung, ist dabei vom Typ `QTextEdit`. Als Teil und Leistung der Qt-Bibliothek verfügt `QTextEdit` über alle wesentlichen Merkmale und Funktionen für eine Textverarbeitung, wie z.B. der Möglichkeit der Texteingabe sowie des Kopierens und Verschiebens von Text. Somit können spätere Texteingaben in der Anwendung editiert werden, und dies ohne dass der Entwickler hierfür eigene Programmierleistung erbringen muss.

Als zentrales Widget füllt `TextEdit` den Fensterbereich zwischen Menü- und Symbolleisten sowie der Statuszeile komplett aus. Als Schrift wird Courier festgelegt.

Sequentiell werden nun die Funktionen `createActions()`, `createMenus()`, `createToolBars()` und `createStatusBar()` aufgerufen. In diesen werden die Aktionen, Menüs etc. definiert und erstellt. Der Aufruf der Funktion `readSettings()` lädt beim Starten der Anwendung die im vorigen Programmlauf zuletzt genutzten Benutzereinstellungen.

Die Signal-Slot-Verbindung erstellt eine Verbindung zwischen dem Dokument in `TextEdit` und dem Slot `documentWasModified()`. Das Signal ist `contentsChanged()`. Ändert sich der Inhalt des Dokumentes wird somit das Signal ausgelöst und der entsprechende Slot aufgerufen. Slots müssen dazu zuvor definiert worden sein (hier in der Headerdatei der Klasse) und später als C++-Funktion implementiert werden. Die Funktionen `connect()`, `SIGNAL()` und `SLOT()` sind dabei Bestandteil der Syntax.

Der Signal-Slot-Mechanismus sowie Aktionen sind wesentliches Element von Qt und bilden die Basis für Qts Plattformunabhängigkeit. Über Signal-Slot-Verbindungen können Objekte miteinander verbunden werden. Slots ähneln normalen C++-Methoden. Der Unterschied besteht darin, dass Slot-Methoden mit Signalen verbunden werden können. In diesem Fall wird die Methode bei jedem Signal automatisch aufgerufen. Dieses Konzept ist plattformunabhängig, denn Slots und Signale werden durch Qt interpretiert und für das jeweils entsprechende System aufbereitet. Qt fungiert quasi als „Vermittler“ zwischen Anwendung und Betriebssystem.

Das Konzept der Aktionen ist mit dem Signal-Slot-Mechanismus eng verbunden. Durch Aktionen vereinfacht Qt die Programmierung von Menü- und Symbolleisten. So handelt es sich bei einer Aktion um ein Element, das sowohl Menü- als auch Symbolleisten hinzugefügt werden kann. Da ein Menüeintrag stets eine Programmfunktion aufrufen soll,

werden Aktionen wie „Neues Dokument“ über Signal-Slot-Verbindungen mit entsprechenden Funktionen verknüpft.

Bild 71 zeigt die allgemeine Definition von Aktionen am Beispiel der Funktion `createActions()`, die vom Konstruktor aufgerufen wird und die Aktionen für die Anwendung erstellt.

Die Aktion wurde vom Verfasser `newAct` genannt und öffnet eine neue Datei z.B. wie es aus Word bekannt ist durch „Datei – Neu“. Die Aktion soll ein Symbol haben (📄), eine Beschriftung („New“), ein Tastaturkürzel (Ctrl+N) und einen erläuternden Hinweis in der Statuszeile. Dies geschieht über Aufrufe entsprechender Qt-Funktionen wie z.B. `setShortcut`.

```
//Methode createActions in Klasse MainWindow, dargestellt durch „:“
void MainWindow::createActions()
{
    //Eine QAction newAct wird erzeugt mit den Übergabewerten
    //Symbol, Beschriftung, aufrufendes Element
    newAct = new QAction(QIcon(":/images/new.png"), tr("&New"), this);
    //Variable newAct wird ein Tastaturkürzel zugewiesen, dies erfolgt
    //über einen Zeiger auf die Qt-Funktion setShortcut
    newAct->setShortcut(tr("Ctrl+N"));
    //newAct wird eine Meldung für die Statuszeile zugewiesen
    newAct->setStatusTip(tr("Create a new file"));
    //Funktion wird für spezielles Signal (triggered) ein Slot
    //(Funktion newFile()) zugewiesen
    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));
}
(...)
```

Abb. 71: Erzeugen von Aktionen, hier „Neues Dokument“

Die vom Benutzer durch „Datei – Neu“ oder durch Klick auf das entsprechende Symbol ausgeführte Aktion löst das Signal `triggered` aus, welches über `connect` die Funktion `newFile()` aufruft. Dies ist eine typische Anwendung einer Signal-Slot-Verbindung.

Mit der Funktion `tr()`, abgekürzt für „*translate*“, werden alle Zeichenketten, die auf der Benutzungsoberfläche angezeigt werden, für die Übersetzung in Fremdsprachen markiert. Dies wird im Rahmen des zweiten Software-Prototyps detailliert vorgestellt.

Die Vorgehensweise bei den anderen Aktionen ist nun entsprechend, wie exemplarisch zwei weitere Aktionen zeigen.


```
(...)
saveAsAct = new QAction(QIcon(":/images/filesaveas.png"),
    tr("Save &As..."), this);
saveAsAct->setStatusTip(tr("Save the document under a new name"));
connect(saveAsAct, SIGNAL(triggered()), this, SLOT(saveAs()));

exitAct = new QAction(tr("E&xit"), this);
exitAct->setShortcut(tr("Ctrl+Q"));
exitAct->setStatusTip(tr("Exit the application"));
connect(exitAct, SIGNAL(triggered()), this, SLOT(close()));
(...)
```

Abb. 72: Weitere Aktionen werden erzeugt

Die so in den Aktionen definierten Slots `newAct`, `exitAct` etc. müssen ebenfalls als Funktionen implementiert werden. Bild 73 zeigt die Funktion `newFile()`. Durch Aufruf der Funktion `maybeSave()` fragt sie zur Sicherheit zunächst, ob die aktuelle Datei bereits gesichert wurde. Wurde die Datei bereits gesichert, wird das Objekt `textEdit` (das Hauptfenster) gelöscht sowie der Dateiname auf einen leeren String zurückgesetzt, also eine neue Datei geöffnet. Ansonsten wird aus der Funktion `maybeSave` heraus zunächst die aktuelle Datei gespeichert.

```
void MainWindow::newFile()
{
    if (maybeSave()) { //Wenn-Abfrage „Wurde Datei gesichert?“
        textEdit->clear(); //Objekt textEdit wird gelöscht
        setCurrentFile(""); //aktueller Dateiname wird auf leer gesetzt
    }
}
```

Abb. 73: Funktion `newFile()`

Bild 74 zeigt die Funktion `closeEvent`, die vom Slot `close` aufgerufen wird. `closeEvent` ist eine Standardfunktion von Qt, die aufgerufen wird, wenn ein Programm geschlossen werden soll, also der Benutzer den Menüeintrag „Datei/Beenden“ auswählt oder in der Titelleiste des Programms das X-Icon zum Schließen wählt oder im Fenstermenü den Eintrag „Schließen“.

In der Funktion `closeEvent`, die also über drei mögliche Signale ausgeführt werden kann, wird sicherheitshalber geprüft, ob die Datei bereits gespeichert wurde (`maybeSave`). Danach werden die letzten Einstellungen für die spätere Wiederbenutzung des Editors mit den letzten Einstellungen des Benutzers gesichert. Abschließend wird das Ereignis

nis (das `closeEvent`) akzeptiert und somit das Programm beendet. Anderenfalls wird das Event verworfen (siehe Bild 74). Die bekannte Sicherheitsabfrage erfolgt innerhalb von `maybeSave`, siehe unten.

```
//Funktion closeEvent in Klasse MainWindow
void MainWindow::closeEvent(QCloseEvent *event)
{
    //Liefert maybeSave true oder false zurück?
    if (maybeSave()) {
        //wenn true, dann wird Funktion WriteSettings() aufgerufen
        //und das Event akzeptiert
        WriteSettings();
        event->accept();
        //wenn false, wird das Event verworfen
    } else {
        event->ignore();
    }
}
```

Abb. 74: Funktion `closeEvent()`

Die Abbildungen 73 und 74 zeigen Aufrufe der Funktion `maybeSave()` die als Boolesche Funktion entsprechend Abb. 75 programmiert wird.

```
//Funktion maybeSave liefert Ergebnis vom Typ „bool“, also true oder false
bool MainWindow::maybeSave()
{
    //Wurde das Objekt textEdit modifiziert?
    if (textEdit->document()->isModified()) {
        //Wenn ja, gib eine MessageBox aus mit den Schaltflächen Yes
        //(Standardschaltfläche), No und Cancel (Escape-Button) und weiteren
        //Eigenschaften
        int ret = QMessageBox::warning(this, tr("Application"),
            tr("The document has been modified.\n"
                "Do you want to save your changes?"),
            QMessageBox::Yes | QMessageBox::Default,
            QMessageBox::No,
            QMessageBox::Cancel | QMessageBox::Escape);
        //Wird Schaltfläche „Yes“ angeklickt, wird Funktion save() aufgerufen
        if (ret == QMessageBox::Yes)
            return save();
        //Wird Schaltfläche „Cancel“ gewählt wird false zurückgegeben
        else if (ret == QMessageBox::Cancel)
            return false;
    }
    //Standardrückgabewert true
    return true;
}
```

Abb. 75: Funktion `maybeSave()`

Die Funktion `maybeSave()` (siehe Bild 75) überprüft dabei, ob das Mainwidget (also das Hauptfenster) der Anwendung, hier `textEdit`, modifiziert wurde. Dazu nutzt sie die Qt-Funktion `isModified()`. Gegebenenfalls wird eine Nachricht angezeigt mit der Meldung an den Benutzer, dass das Dokument geändert und noch nicht gespeichert wurde. Ähnlich wie in VBA in der Funktion `MsgBox` kann der Programmentwickler das Erscheinungsbild des Dialogfensters bestimmen, z.B. welche Schaltflächen angezeigt werden. Wählt der Benutzer „Ja“ wird die Funktion `save()` aufgerufen und das Dokument gespeichert, wählt er „Cancel“ werden die Änderungen verworfen.

Erst diese vielschichtige Programmierung des Anwendungsentwicklers mit C++ und Qt-Funktionen erzeugt das Programm, das der Benutzer bei der Ausführung wahrnimmt: Die Klickfolge auf „File/New“ löst das Signal aus, welches die Funktion `newFile()` aufruft. Diese wiederum ruft die Funktion `maybeSave()` auf, die dem Benutzer die Frage stellt, ob er die Änderungen speichern möchte. Je nach Antwort des Benutzers wird nun entweder die Datei gespeichert oder die Änderungen verworfen. Danach springt das Programm zurück in `newFile()`, löscht den Inhalt von `textEdit` und setzt eine leere Zeichenkette als aktuellen Dokumentnamen.

Entsprechend werden die noch fehlenden Slots und die zugehörigen Funktionen implementiert, z.B. für das Speichern und Öffnen einer Datei oder für das Laden der Voreinstellungen. Dementsprechend müssen weitere Funktionen angelegt werden, z.B. `documentWasModified()`, welche überprüft, ob das Dokument geändert wurde, oder `setCurrentFile()`, welche den aktuellen Dateinamen extrahiert und in der Titelzeile des Programms anzeigt.

Bislang wurden lediglich Aktionen definiert und die entsprechenden Funktionen implementiert. Nun müssen diese Aktionen in die Benutzungsoberfläche eingefügt werden. Dies geschieht über die Funktionen `createMenus()` und `createToolBars()`, die im Konstruktor aufgerufen werden. Dabei wird das Erscheinungsbild der Menüs und Symbolleisten durch Qt erzeugt und entspricht somit automatisch dem gewohnten „Look & Feel“ des verwendeten Betriebssystems.

Abb. 76 zeigt die Funktion `createMenus()`. Sie erzeugt die Hauptmenüs „File“, „Edit“ und „Help“ sowie die entsprechenden Untermenüs. Dabei wird im Menübalken des Me-

nüs der Name des Menüs angezeigt („File“). In den folgenden Zeilen werden die jeweiligen Aktionen diesem Menü zugeordnet. Alle weiteren Informationen sind bereits in der Aktion definiert (Text, Icon, Tastaturkürzel etc.) und dürfen an dieser Stelle nicht wiederholt werden.

```
//Funktion createMenus() ohne Rückgabewert
void MainWindow::createMenus()
{
    //menuBar() initialisiert eine QMenuBar, addMenu fügt erstes
    //Hauptmenü hinzu
    fileMenu = menuBar()->addMenu(tr("&File")); //Hauptmenü „Datei“
    //addAction fügt fileMenu Aktionen hinzu
    fileMenu->addAction(newAct); //Menüpunkt „Neu“
    fileMenu->addAction(openAct); //Menüpunkt „Öffnen“
    fileMenu->addAction(saveAct); //Menüpunkt „Speichern“
    fileMenu->addAction(saveAsAct); //Menüpunkt „Speichern
    //unter“

    //ein Trennstrich wird eingefügt
    fileMenu->addSeparator();
    fileMenu->addAction(exitAct);

    //Zweites Hauptmenü editMenu wird erstellt
    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAct);
    editMenu->addAction(copyAct);
    editMenu->addAction(pasteAct);

    //Ein Trennstrich auf Hauptmenüebene
    menuBar()->addSeparator();

    helpMenu = menuBar()->addMenu(tr("&Help"));
    helpMenu->addAction(aboutAct);
    helpMenu->addAction(aboutQtAct);
}
```

Abb. 76: Funktion createMenus()

Vor dem Eintrag des Hilfe-Menüs helpMenu findet sich eine Besonderheit: addSeparator. Dies bewirkt bei einigen Betriebssystemen (u.a. Unix oder auch Mac OS), dass das Hilfemenü nicht fortlaufend linksbündig, sondern an den rechten Rand der Menüleiste platziert wird. Der Separator trennt somit die Menüleiste in zwei Bereiche, wobei der erste links und der zweite rechts in der Menüleiste ausgerichtet wird. Wird für verschiedene Betriebssysteme entwickelt, sollte dieser Eintrag berücksichtigt werden. Bei der Entwicklung für nur ein System, welches das Hilfemenü nicht separat anordnet, z.B. Windows, kann darauf verzichtet werden.

Auf die gleiche Art wie die Menüs werden auch die Symbolleisten programmiert, siehe Abb. 77.

```
//Funktion createToolBars() ohne Rückgabewert
void MainWindow::createToolBars()
{
    //addToolBar fügt eine neue Symbolleiste fileToolBar hinzu
    fileToolBar = addToolBar(tr("File"));
    //addAction fügt fileToolBar eine Aktion hinzu
    fileToolBar->addAction(newAct);
    fileToolBar->addAction(openAct);
    fileToolBar->addAction(saveAct);

    //Zweite Symbolleiste editToolBar wird eingefügt
    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(cutAct);
    editToolBar->addAction(copyAct);
    editToolBar->addAction(pasteAct);
}
```

Abb. 77: Erstellen der Symbolleiste

Auch hier übernimmt Qt die Darstellung der entsprechenden Eigenschaften auf dem Bildschirm.

Als letztes Element einer zwar noch klassischen, aber zumindest symbolunterstützten Benutzungsoberfläche wird die Statuszeile vorgestellt, ebenfalls elementarer Teil einer modernen Benutzungsoberfläche. Die Qt-Funktion `createStatusBar()` wird dazu im Konstruktor aufgerufen.

```
void MainWindow::createStatusBar()
{
    //Element statusBar wird per Zeiger referenziert
    statusBar() ->showMessage(tr("Ready"));
}
```

Abb. 78: Erstellen der Statuszeile

Die Statuszeile befindet sich üblicherweise am Fuß des Anwendungsfensters und wird dort durch Qt beim Initialisieren der Anwendung ausgegeben. Die erste Meldung „Ready“ wird durch Meldungen der nachfolgenden Aktionen ersetzt.

Ein weiteres Element moderner Benutzungsoberflächen ist die Änderung der Gestalt des Mauszeigers, um dem Benutzer Programmzustände mitzuteilen. Auch dies ist vom Entwickler jeweils an entsprechender Stelle zu programmieren.

So zeigt z.B. ein „Sanduhr“-Cursor an, dass das System ausgelastet ist und momentan keine Eingaben verarbeitet werden können. Der Sanduhr-Cursor ist ein weltweit verständliches Symbol, also textfrei ohne Übersetzungsbedarf. Bild 79 zeigt, wie eine solche Funktion zu implementiert ist.

```
(...)  
//Schreiben des Text in eine Datei  
QTextStream in(&file);  
//Funktion setOverrideCursor ersetzt aktuellen Cursor mit Qt::WaitCursor  
QApplication::setOverrideCursor(Qt::WaitCursor);  
textEdit->setPlainText(in.readAll());  
//nachdem Datei geschrieben wurde wird der vorherige  
//Cursor wieder hergestellt  
QApplication::restoreOverrideCursor();  
(...)
```

Abb. 79: Implementieren einer Cursoränderung

Während die Datei von der Festplatte geladen wird (`QTextStream in(&file)`) erscheint der auf dem System gültige `waitCursor` auf dem Bildschirm. Wenn diese Operation beendet ist, wird der ursprüngliche Cursor mit `restoreOverrideCursor()` wieder hergestellt.

Diese Funktion sollte für alle Situationen programmiert bzw. aufgerufen werden, in denen das System so ausgelastet sein könnte, dass Eingaben des Benutzers nicht sofort verarbeitet werden.

Damit ist der Funktionsumfang des Prototyps einer klassischen, aber Styleguide-konformen, verständlichen Benutzungsoberfläche programmtechnisch nachgewiesen, siehe Bild 64.

Das Beispiel zeigt, wie das innovative Signal-Slot-Konzept mit Aktionen den Entwickler bei der Programmierung von styleguidekonformen Menüs und Symbolleisten unterstützt. Während Signale Nachrichten an einen Slot senden, was durch Qt bzw. Windows zur Laufzeit automatisch überwacht wird, müssen die Funktionen, wie etwa das Neuanlegen einer Datei oder das Speichern durch den Anwendungsprogrammierer definiert werden.

Selbst Standardfunktionen zeitgemäßer Benutzungsoberflächen, wie etwa das Erkennen von Änderungen und der dann übliche Hinweis zum Speichern von ungesicherten Änderungen, muss bei diesem programmtechnischen Ansatz vom Entwickler individuell programmiert werden. Dennoch bieten nur Programmiersprachen wie C++ und Klassenbibliotheken wie Qt dem Entwickler die Möglichkeit, eigenständige Software zu entwickeln. Entwicklungsumgebungen wie VBA liefern immer nur Ergebnisse, die innerhalb der Entwicklungsumgebung lauffähig sind und eignen sich daher nicht für die Zwecke dieser Arbeit.

Dabei zeigt der in diesem Prototyp vorgestellte programmtechnische Ansatz symbolunterstützte Benutzungsoberflächen zu gestalten, dass dies bei den Weltmarktführern für CAD-Hochleistungssysteme noch nicht verwirklicht wurde, wie die Untersuchungen in Kapitel 2 gezeigt haben.

5.3 Software-Prototyp II: Automatische Übersetzung von unvermeidbaren Texten

Der erste Softwareprototyp führte den Nachweis für klassische Benutzungsoberflächen, bei denen Anwendungssoftware nicht ohne landessprachlichen Text auskommt.

Die sprach- und textfreie Gestaltung von Benutzungsoberflächen für Bausoftware, wie sie in dieser Arbeit gefordert und im programmtechnischen Ansatz entwickelt wird, stößt an ihre Grenzen. Es verbleiben neben Zahlen auch einige wenige Texte, auf die nicht verzichtet werden kann. Im programmtechnischen Ansatz gilt es dann nicht nur einen möglichst effektiven Weg automatischer Übersetzung zu finden, sondern auch die Verschiedenheiten aller internationalen Zeichensätze zu beachten.

Dazu seien zunächst die Hintergründe und Probleme von internationaler Software erörtert.

Früher war es durch einen auf 7 bzw. 8 Bit begrenzten Zeichenvorrat (maximal 128 bzw. 256 Zeichen) nur möglich, den für ein spezielles Land notwendigen Zeichensatz zu speichern, z.B. der Zeichensatz ASCII oder ISO 8859-1 (Latin 1), der in Europa Verwendung findet und u.a. auch die deutschen Umlaute beinhaltet [Blanc01]. Anstelle dieser interna-

tional bislang inkompatiblen Codierungen wurde Unicode mit einem Zeichenvorrat von 1.114.112 ($=2^{20}+2^{16}$) Zeichen [WIKI-Unicode] als internationaler Standard festgelegt. Unicode enthält die Zeichensätze aller Länder unter Beibehaltung der ursprünglichen Codepositionen. Dies ermöglicht ein Konvertieren in verschiedene „Codepages“, z.B. von Deutsch (Codepage Latin Nr. 1252) zu Persisch (Codepage Arabisch/Farsi Nr. 1256).

Die `QString`-Klasse von Qt nutzt Unicode als internationale Basis und speichert Zeichenketten in diesem Zeichenformat ab. So unterstützt Qt auf den unterschiedlichen Betriebssystemen alle westlichen Sprachen sowie alle asiatischen, griechischen, kyrillischen sowie die arabischen Sprachen, zu denen auch Persisch zählt.

Qt beachtet sogar laut [Qt-Doc] die bei diesen Zeichensätzen besonderen Eigenschaften, wie z.B. Schreibrichtung, Zeilenumbrüche und Ligaturen, d.h. die typografische Verbindung zweier Buchstaben zu einem. Die wohl bekannteste deutsche Ligatur ist das „ß“ (Eszett).

Unter Beachtung dieser Vorüberlegungen wird nun der Prototyp I erweitert zu einer mehrsprachigen Benutzungsoberfläche als grundsätzliches Beispiel für unverzichtbare Texte einer Benutzungsoberfläche.

Dazu werden im ersten Schritt für klassische Benutzungsoberflächen die Menüeinträge etc. automatisch in eine andere Sprache übersetzt.

In einem zweiten Schritt wird, auf den dadurch gewonnen Erkenntnissen aufbauend, ein neuer, innovativer Ansatz entwickelt, wie vollkommen textfreie Menüs mittels der vorgestellten Qt-Werkzeuge zu realisieren sind. So bildet diese Arbeit auch einen Teil des experimentellen Nachweises für [Azad07].

Die programmtechnische Realisierung automatischer Übersetzung erfolgt in Qt in drei Schritten:

- 1) Im Entwicklungswerkzeug VisualStudio wird über den Menüeintrag „Qt – Create New Translation File“ eine „Translation Source“ als `ts`-Datei angelegt. Alle Texte (Zeichenfolgen), die im gesamten Quellcode (wie im Prototyp I bereits gezeigt) als Parameter der `tr()`-Aufrufe genannt sind, werden gesucht und in dieser Datei im XML-Format gespeichert.

- 2) Die `ts`-Datei wird mit dem Qt-Werkzeug QtLinguist geöffnet und die Einträge von Fachübersetzern übersetzt.
- 3) Aus der `ts`-Datei wird über „File – Release“ im QtLinguist eine `qm`-Datei (Qt Message) erzeugt, die in der Anwendung über `QTranslator` geladen wird.

Die Schritte 1 und 3 werden von den Anwendungsentwicklern durchgeführt, Schritt 2 ist Fachübersetzern vorbehalten.

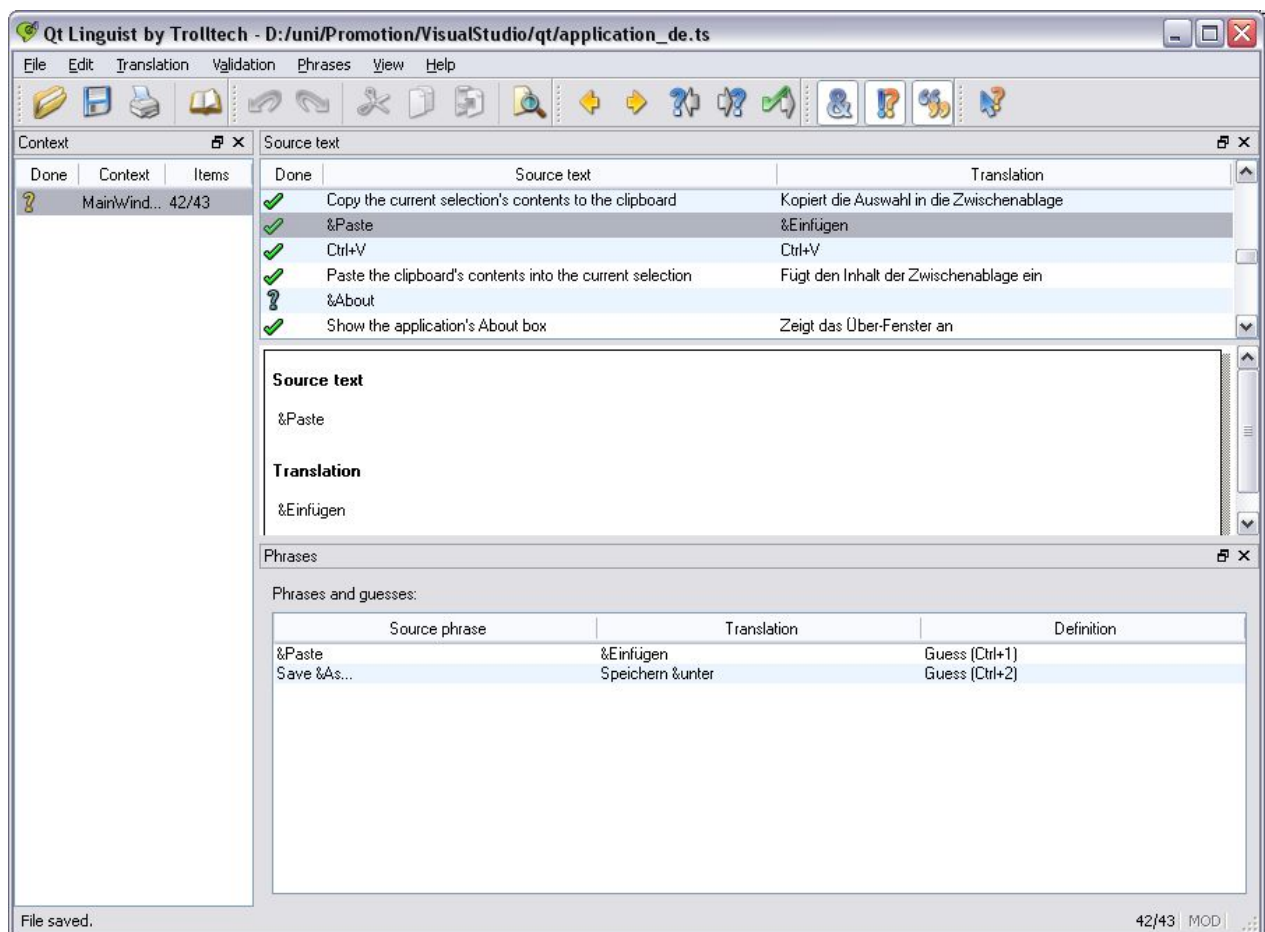


Abb. 80: Qt-Übersetzungswerkzeug QtLinguist, Übersetzung Englisch/Deutsch

Im zuvor vorgestellten Prototyp I wurden bereits alle Zeichenketten, die zur Benutzungsoberfläche gehören, vorsorglich für die automatische Übersetzung als Parameter in `tr()`-Aufrufe eingefasst. QtLinguist extrahiert alle `tr()`-Funktionsaufrufe des Quellcodes, wodurch die Übersetzungsdatei erstellt wird. Wie in Bild 80 zu sehen, gestaltet sich dadurch die Arbeit für den Fachübersetzer sehr systematisch.

Bild 80 zeigt die im QtLinguist geöffnete Übersetzungsdatei mit deutscher Übersetzung, Bild 81 mit persischer Übersetzung. Im linken Rahmen werden automatisch alle Objekte der Benutzungsoberfläche angezeigt, in denen Elemente zur Übersetzung gefunden wurden. Hier wurde also die Klasse „MainWindow“ mit 43 Elementen gefunden, die Teil der Benutzungsoberfläche sind. Im rechten, oberen Fenster, werden unter „Sourcetext“ alle gefundenen Zeichenketten aufgelistet. Ein vorangestelltes Fragezeichen bzw. ein OK-Häkchen signalisieren, ob die Übersetzung abgeschlossen oder noch offen ist. Die Übersetzung wird im mittleren Fenster eingegeben. Im unteren Fenster schlägt Qt dem Übersetzer mögliche Übersetzungen vor, jedoch ohne Fachkompetenz.

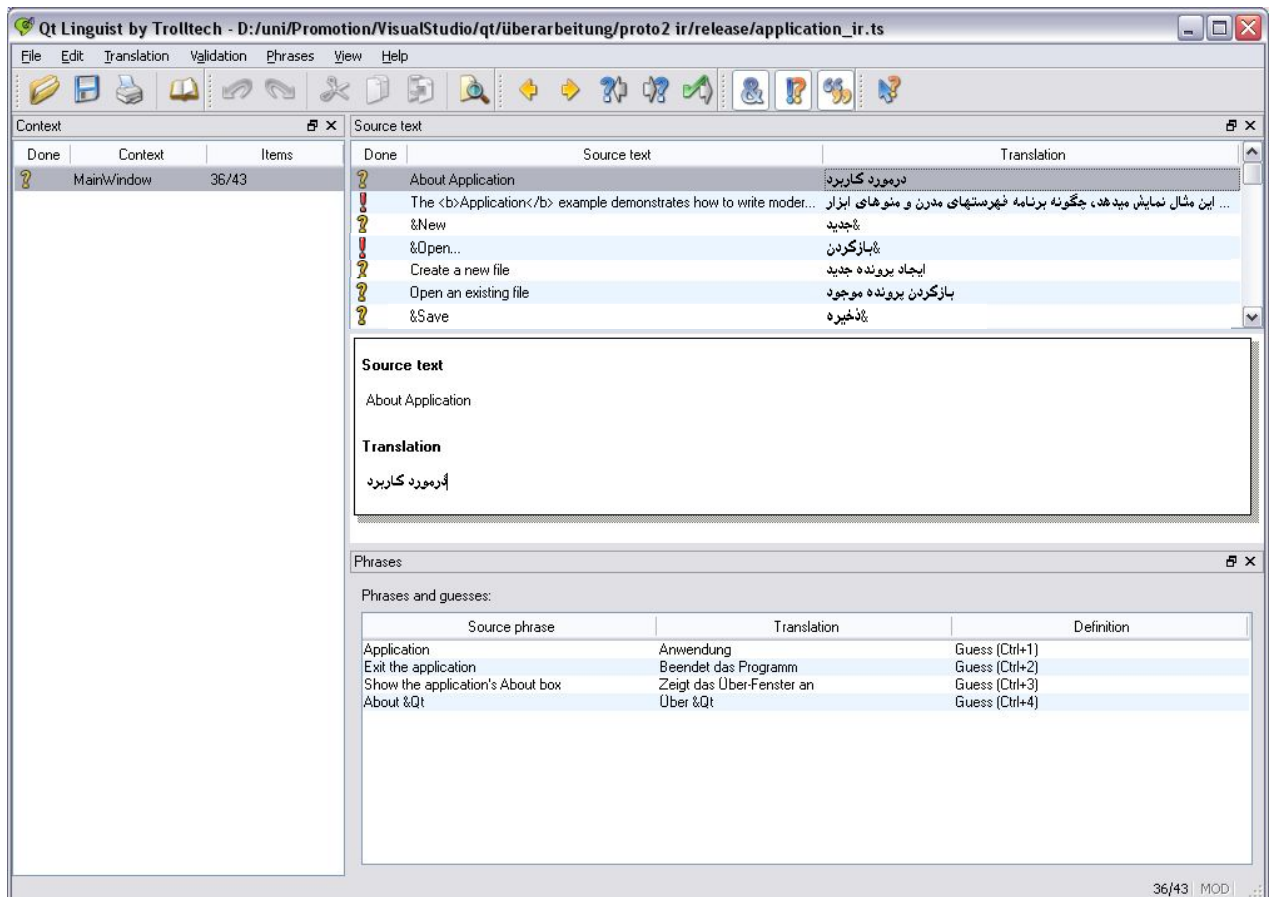


Abb. 81: Qt-Übersetzungswerkzeug QtLinguist, Übersetzung Englisch/Persisch

Über den Menüeintrag „File – Release“ im QtLinguist wird die übersetzte Version freigegeben und als `qm`-Datei gespeichert.

```
#include <Qapplication>
//Inkludieren der Qtranslator-Bibliothek
#include <QTranslator>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    Q_INIT_RESOURCE(application);

    QApplication app(argc, argv);

    //Erzeugen eines neuen QTranslator-Objektes
    QTranslator translator;
    //Laden der Übersetzungsdatei
    translator.load("translation");
    app.installTranslator(&translator);

    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

Abb. 82: Implementierung der Übersetzungsdatei (graue Textbereiche unverändert zu Abb. 67)

Die Übersetzungsdatei wird entsprechend Bild 82 in das Anwendungsprogramm integriert. Dazu dient ein Objekt vom Typ `QTranslator`, welches die Übersetzungsdatei bei Programmstart lädt. Welche Übersetzungsdatei geladen wird, legt der Anwender nach seinem persönlichen Bedarf z.B. bei der Installation des Programms selbst fest. Durch den im Quellcode allgemeingehaltenen Dateinamen (z.B. `translation`) kann so jede Übersetzung dynamisch geladen werden. Diese Lösung, wie Software international vertrieben wird, bildet die Basis für Kooperationen von internationalen Projektteams in weltweiten Netzen bei denen jeder Benutzer die Anwendung in der Sprache seines Landes nutzt.

Ebenso können auf diese Weise mehrere Sprachen parallel installiert werden und die entsprechende Sprachversion beim Programmstart durch den Benutzer ausgewählt werden. Diesen Ansatz verfolgt z.B. Tekla bei Tekla Structures. Hier kann der Benutzer z.B. „Tekla Structures 11.2 deutsch“ ebenso wie „Tekla Structures 11.2 english“ auswählen und starten.

Den experimentellen Nachweis dieses erweiterten Ansatzes zeigt Bild 83.

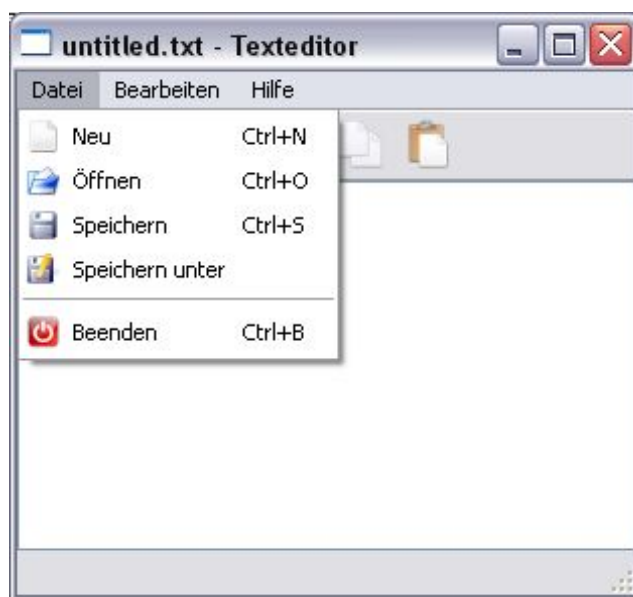


Abb. 83: Deutsch lokalisierte, klassische Benutzeroberfläche

Dieses Programmbeispiel zeigt somit, wie Anwendungen mit Qt lokalisiert, also um weitere Landessprachen erweitert werden können. Die Behandlung von unverzichtbaren Texten in Anwendungen wird dadurch so vereinfacht, dass sie auch für kleine Softwarebüros wirtschaftlich aussichtsreich wird.

Basierend auf den bereits vorgestellten Qt-Werkzeugen wird nun ein neuer, innovativer Ansatz entwickelt, Menüs sprach- und textfrei nur durch Symbole zu realisieren. Bild 84 zeigt die entwickelte Lösung zu der nachfolgend der programmtechnische Ansatz entwickelt wird.

Erreicht wird dieses Ziel überraschend einfach mit den vorgestellten Werkzeugen durch Anpassung der Übersetzungsdatei. Dazu werden im QtLinguist die gefundenen Zeichenketten der Menübezeichnungen und der Tastaturkürzel durch jeweils ein Leerzeichen ersetzt. Das Ergebnis ist ein Menü, welches nur noch aus den in den Aktionen definierten Symbolen besteht, sowie den nicht sichtbaren Leerzeichen.



Abb. 84: Benutzungsoberfläche mit sprachfreien Menüs

Diese Lösung ist in dieser Form bislang einzigartig und auf dem Weltmarkt noch nicht umgesetzt. Sie bildet den experimentellen Beweis für [Azad07], der in seiner Arbeit eine weitgehend auf Symbolen und Anfassern basierte CAD-Benutzungsoberfläche entwickelt.

Der dynamische Sprachwechsel, also vom Benutzer während der Benutzung der Anwendung, wird von Qt nicht automatisch unterstützt. In [Blanc01] wird eine durch den Anwendungsentwickler manuell zu implementierende Lösung beispielhaft vorgestellt: Anstatt die Übersetzungsdatei wie gezeigt im Hauptprogramm statisch zu laden, muss dem Benutzer ein Menü angeboten werden, die Sprache zu wechseln. Zusätzlich muss eine separate Funktion alle im Programm enthaltenen Zeichenketten zur Laufzeit ersetzen und aufrufen, wenn das Programm gestartet ist und der Benutzer die Sprache ändert.

Dabei übernimmt die Funktion `retranslateStrings()` die Aufgabe, die Zeichenketten in der entsprechenden Sprache in die Benutzungsoberfläche einzufügen. So werden textliche Beschriftungen in Menü- oder Symbolleisten nicht mehr in den Aktionen implementiert, sondern in `retranslateStrings()`, ebenso wie alle weiteren `tr()`-Aufrufe der Klasse `MainWindow`.

Wählt der Benutzer im Sprachmenü eine andere Sprache aus, wird die mit dieser Aktion verbundene Funktion `switchToLanguage()` aufgerufen, die die entsprechende Übersetzungsdatei lädt und danach `retranslateStrings()` aufruft. Dadurch werden die Beschriftungen der Benutzungsoberfläche geändert und direkt angezeigt.

Diese Lösung ist aufgrund der fehlenden automatischen Unterstützung durch Qt komplexer und muss vom Anwendungsentwickler manuell entwickelt werden. Der entsprechende Quellcode für diese erweiterte Lösung des Software-Prototypen II befindet sich im Anhang dieser Arbeit.

Eine weitere Problematik der landestypischen Darstellung stellt sich beim internationalen Austausch von Zeichnungen und Dateien. So soll eine mit deutschen Texten und Ziffern beschriftete Bauzeichnung beim Öffnen in einem anderen Land mit dem dort gültigen Zeichensatz angezeigt werden, um dort sofort verstanden zu werden.

Ein typisches Beispiel aus der Baupraxis zeigt Bild 85 mit einer Einzelteilzeichnung in deutscher Sprache und Bild 86 mit derselben Zeichnung in Persisch.



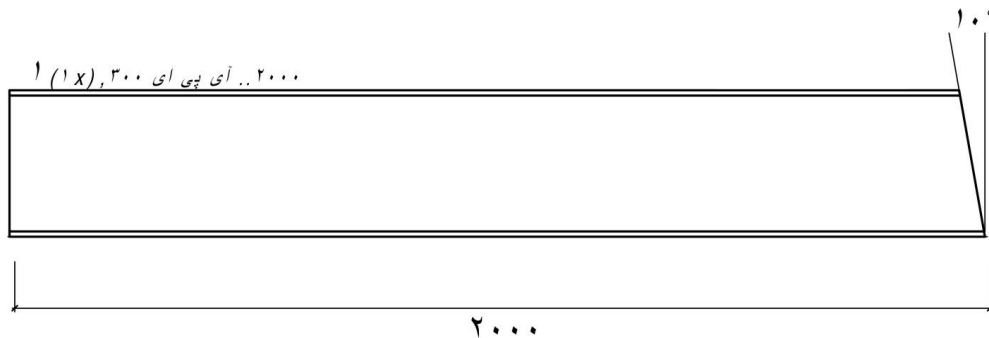
**** Stückliste für eine Liefer- Pos. 1 TOTAL 1 x Ausführen ****

Nr.	Tnr	Pos.	Benennung	Stk.	Profil	Material	Länge	Gewicht	Bem.
1	1	1	STUETZE	1	IPE300	S235JR2	2000	84.40	

Abb. 85: Einzelteilzeichnung, Beschriftung und Maßzahlen in deutsch

Als programmtechnische Lösung für dieses Problem bietet sich XML als Dateiformat an. XML steht für „Extensible Markup Language“ und ist nach [WIKI-XML] ein Standard zur Modellierung von strukturierten Daten, d.h. XML definiert Regeln für den Aufbau von Do-

kumenten, die so mit formal gleichen, inhaltlich aber unterschiedlichen Inhalten zum internationalen Austausch geeignet sind. Dabei können XML-Dateien gleichzeitig Datenstrukturen wie Text und Grafik enthalten. XML-Dateien sind hierarchisch in Form einer Baumstruktur geordnet. Grundgedanke ist die Trennung der Daten und ihrer Repräsentation.



****فهرست اجزاء، جزء ۱، تعداد کل ۱ × تهیه شود****

توضیحات	وزن کیلوگرم	طول	مشخصات مکانیکی	پروفیل	تعداد	نام قطعه	جزء	آی ان ار شماره
	۸۴,۴۰	۲۰۰۰	S235JRG2	PE300	۱	ستون	۱	۱

Abb. 86: Einzelzeichnung, Beschriftung und Maßzahlen in Persisch

Mit diesem Ansatz sind technische Dokumente in elektronischer Form je nach Anwender in weltweiten Netzen in jeweils landesspezifischer Form auf dem Bildschirm oder Drucker auszugeben.

Über die zu definierende, international festzulegende Notation (d.h. Festlegung der XML-Strukturelemente) kann nun ein Parser beim Öffnen der Zeichnung die entsprechenden Attribute anzeigen und so die textlichen Zeichnungselemente übersetzen.

Qt bietet keine Werkzeuge, die dafür geeignet sind. Diese Entwicklung eines solchen XML-basierten, neuen Datenformats ist von Entwicklern aktueller CAD-Systeme zu vollziehen und in diese zu implementieren. So erzielt man ein universelles Datenaustauschformat, welches zudem die Sprachbarrieren erfolgreich überwindet.

Dass derartige Innovationen im Bauwesen möglich und durchsetzbar sind, hat in den letzten Jahren der Deutsche Stahlbauverband bewiesen. Seine weltweit kopierte Stücklisten- und NC-Schnittstelle wurde ganz in diesem Sinne auf XML umgestellt.

5.4 Software-Prototyp III: Elementares CAD-System mit sprach- und textfreier Benutzungsoberfläche

In diesem Software-Prototypen wird exemplarisch die Grafikprogrammierung für textfreie, grafische Benutzungsoberflächen mit Qt nachgewiesen, soweit mit diesem Werkzeug möglich.

Dieser Prototyp gestattet, im Prinzip ähnlich zur interaktiven Platzierung und Änderung von Profilstäben in einem CAD-System für das Bauwesen, das Platzieren und Verändern von Rechtecken und Linien als modellhafte Vereinfachung, siehe Bild 87. Diese Elemente sollen möglichst sprach- und textfrei interaktiv manipulierbar sein.

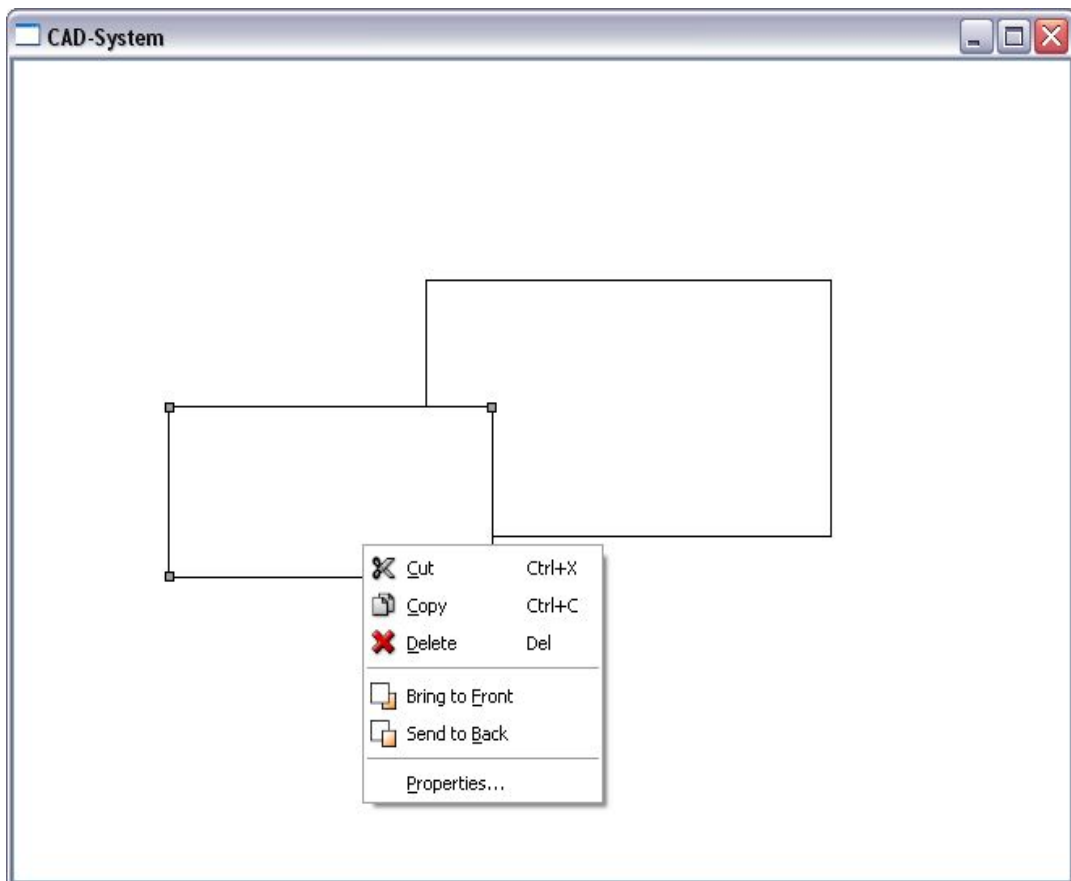


Abb. 87: CAD-System mit klassischer, symbolunterstützter Benutzungsoberfläche

Dabei wird die Komplexität der Grafikprogrammierung deutlich. Neben der Definition einer Klasse für das Anwendungsfenster werden für jedes Grafikelement, welches darge-

stellt werden soll, weitere Klassen benötigt. Ebenso müssen Mausereignisse laufend abgefragt und auf diese mit entsprechenden Ereignisprogrammen reagiert werden.

Im einführenden Software-Prototyp I war `QTextEdit` die Basisklasse und stellte als Qt-Klasse alle elementaren Funktionen für die Anwendung zur Verfügung. Sie ist also so vollständig, dass in der Programmierung nur Aktionen für Menü- und Symbolleisten erstellt werden mussten.

Für Anwendungen mit grafischen Elementen, die der Benutzer wie in CAD-Systemen manipulieren kann, ist die wesentlich weniger konkret mit Leistungen ausgestattete Basisklasse `QCanvas` der Qt-Bibliothek notwendig. Die Möglichkeiten, die diese Klasse bietet, sind durch die Bedeutung des Wortes „Canvas“ (Leinwand) angedeutet: `QCanvas`-Objekte können Elemente beliebiger Form enthalten, die sich auf ihr bearbeiten lassen. Das Verhalten der Objekte ist von Qt naturgemäß nicht vorgebar und muss tief bis ins Detail vom Anwendungsentwickler programmiert werden.

Es gibt nur sehr elementare vordefinierte Klassen, die als Canvas-Elemente zur Verfügung stehen. In diesem Beispiel werden `QCanvasRectangle` und `QCanvasLine` verwendet, aus denen eigene Klassen abgeleitet werden, um die Funktionen der Basisklassen zu erweitern. Diese können aufgrund dieser Programmierung als Grafikobjekte vom Benutzer im Arbeitsfenster platziert und dimensioniert, nachträglich verändert sowie in die Zwischenablage kopiert und aus dieser eingefügt werden.

Hilfreich für die Entwicklung dieses Software-Prototypen waren Untersuchungen von [Blanc01], die die Machbarkeit von grafischen Benutzungsoberflächen mit C++ und Qt-Bibliotheken beweisen, wenn gleichzeitig die entwickelte Software auf verschiedenen Betriebssystemen lauffähig sein soll.

Es zeigt allerdings die Grenzen von Qt für die Forschung auf. Denn das vorgestellte Beispiel belegt, dass die zur Lösung der elementaren Aufgabenstellung dieser Arbeit ausschlaggebenden Anfasser zur direkten Manipulation von Objekten nicht mit der Qt-Bibliothek erzeugt werden können.

Das Fehlen einer entsprechenden seit November 2006 im Konkurrenzprodukt .NET implementierten Funktion wird nach einem intensiven Austausch über dieses Problem auch von Trolltech bestätigt.

Eben dieses Fehlen einer solchen elementaren Funktion zur Unterstützung des Anwendungsentwicklers in Qt war den Entwicklern dort bisher nicht bewusst. Diese Arbeit konnte daher wichtige Impulse an die Entwickler von Qt geben, eine solche Funktion bzw. Klasse zu entwickeln und in zukünftigen Versionen von Qt entsprechend zu implementieren.

Denn eben solch vordefinierten Klassen und ihre entsprechenden Funktionen ermöglichen es den Nutzern einer Klassenbibliothek wie z.B. Qt, diese elementaren Grundfunktionen einer modernen Benutzungsoberfläche nicht selbstständig und individuell programmieren zu müssen. Ein wesentlicher Aspekt modernen Programmierens, der für mehr Effizienz bereits in der Entwicklung sorgt.

Ein vierter Software-Prototyp wird daher aufzeigen, wie die für Effizienz und Verständlichkeit ausschlaggebende direkte Manipulation mittels Anfassern in .NET so implementiert ist, dass mit dem programmtechnischen Ansatz dieser Arbeit als Musterlösung auch CAD-Hochleistungssysteme durch entsprechendes Re-Engineering für die Zukunft des Weltmarktes ertüchtigt werden können.

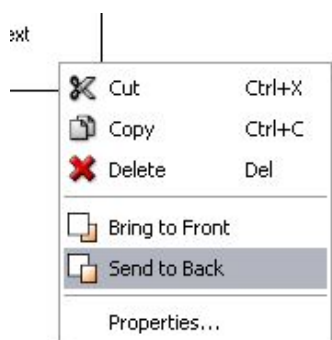


Abb. 88: Klassisches Kontextmenü

Die Aufgabenstellung: Nach Start der Anwendung öffnet ein Rechtsklick in den leeren Bereich des Anwendungsfensters ein Kontextmenü zur Konstruktion von Rechtecken oder Linien (siehe Bild 87).

Ein Doppelklick auf ein eingefügtes Rechteck gibt dem Benutzer die Möglichkeit, einen Beschreibungstext einzugeben, während ein Rechtsklick auf ein vorhandenes Objekt das Kontextmenü mit Menüpunkten zum Kopieren und Bearbeiten anzeigt. Bild 88 zeigt den implementierten, klassischen textbasierten Ansatz aus der bekannten Welt. [Azad07] entwickelt in seiner Arbeit einen entsprechend textfreien Ansatz mit intuitiv verständlichen Symbolen, der mit den in dieser Arbeit entwickelten programmtechnischen Erkenntnissen umgesetzt werden kann.

Aktive Elemente sollen durch kleine Rechtecke in den Ecken als markiert angezeigt werden. Die mustergültige intuitive Lösung, das Objekt über diese Anfasser direkt in der Größe verändern zu können, konnte mit Qt in diesem Software-Prototypen nicht umgesetzt werden.

Dennoch soll im Folgenden gezeigt werden, wie weit die Forschung an dieser Stelle getrieben werden konnte.

Analog zum ersten Software-Prototypen muss zunächst die Klasse definiert werden, die vom Hauptprogramm als `MainWidget` gesetzt wird. Dies ist hier die Klasse `CadView`. Sie stellt den Anwendungsrahmen sowie die wesentliche Anwendungs-Funktionalität zur Verfügung. Wie bereits erläutert, wird die Klasse in der Headerdatei `cadview.h` definiert und in `cadview.cpp` implementiert.

`CadView` wird von `QCanvasView` abgeleitet und erbt somit alle wesentlichen Eigenschaften zur Darstellung von weiteren Elementen (`CanvasItems`). Bild 89 zeigt die Definition der Klasse `CadView`. Die Definitionen der Slots sowie der Methoden und Attribute im `private`-Teil sind dabei analog zum Software-Prototyp 1. Im `protected`-Bereich fallen vier Methoden auf, die auf die Mausereignisse reagieren. Eine detaillierte Beschreibung erfolgt jeweils bei der Implementierung der Funktion.

Ebenfalls in `cadview.h` werden die Klassen der beiden Canvas-Elemente `DiagramBox` und `DiagramLine` definiert.

```

//Klasse CadView erbt alle Eigenschaften von QcanvasView, formal
//gelöst durch „:“
class CadView : public Q3CanvasView
{
//Klasse definiert ein Q_OBJECT
    Q_OBJECT
//Definition des Konstruktors zur Erzeugung von Instanzen dieser Klasse
public:
    CadView(Q3Canvas *canvas, QWidget *parent = 0,
            const char *name = 0);

//Slots für Aktionen definieren
public slots:
    void cut();
    void copy();
    (...)

//Definition der Methoden, die entsprechend des übergebenen Signals
//für die zugehörigen Mausereignisse aufgerufen werden
protected:
    void contentsContextMenuEvent(QContextMenuEvent *event);
    void contentsMouseEvent(QMouseEvent *event);
    void contentsMouseMoveEvent(QMouseEvent *event);
    void contentsMouseEvent(QMouseEvent *event);

private:
    void createAction();
    void addItem(Q3CanvasItem *item);
    (...)

```

Abb. 89: Definition der Klasse CadView in cadview.h

DiagramBox erbt von QCanvasRectangle, welches lediglich ein Rechteck ausgibt. Bild 90 zeigt die Erweiterung der Klasse, um in einem Rechteck einen Text anzuzeigen sowie kleine Quadrate an den Ecken, die ein aktives Element markieren. Dies ist, wie das Programmbeispiel zeigt, mit Qt einfach zu lösen.

Die Möglichkeit aber, diese Anfasser „greifbar“ zu machen, um mit diesen die Größe des Objektes direkt ändern zu können, ist mit der Qt-Bibliothek programmtechnisch nicht lösbar. Wie ein entsprechender Ansatz in .NET gelöst und in Bau-Softwareprodukten eingesetzt werden kann, wird der Software-Prototyp IV aufzeigen.

Der Name der aufgerufenen Funktion `rtti()` leitet sich von „Runtime Type Identification“, also Typinformation zur Laufzeit, ab. Die hier festgelegte RTTI-Konstante 1001 ist willkürlich gewählt und wird später zur Zuordnung des Canvas-Elementes verwendet. Dementsprechend darf jede Konstante nur eindeutig in einer Anwendung genutzt werden. Der `DiagramLine` wird folglich eine RTTI-Konstante von 1002 zugewiesen.

```

//Ableiten der Klasse DiagramBox von der Basisklasse Q3CanvasRectangle
class DiagramBox : public Q3CanvasRectangle
{
public:
//RTTI Variable zur späteren Identifikation
    enum { RTTI = 1001 };

//Konstruktor zur Erzeugung neuer Objekte
    DiagramBox(Q3Canvas *canvas);
//Destruktor zum Löschen des Objekts
    ~DiagramBox();

//Definition von Methoden und Attributen und deren Rückgabewerte:
    void setText(const QString &newText);
    QString text() const { return str; }
    void drawShape(QPainter &painter);
    QRect boundingRect() const;
    int rtti() const { return RTTI; }

private:
    QString str;
};

```

Abb. 90: Klassendefinition für das Canvas-Elemente Rechteck

Auch die Klasse `DiagramLine` ist ein Canvas-Element, welches die Fähigkeit, eine Linie darzustellen, von `QCanvasLine` erbt. Zusätzlich wurde die Klasse erweitert, um eine aktive Linie durch kleine Quadrate am Anfangs- und Endpunkt zu markieren, siehe Bild 91.

```

//Klassendefinition analog zu DiagramBox
class DiagramLine : public Q3CanvasLine
{
public:
    enum { RTTI = 1002 };
    DiagramLine(Q3Canvas *canvas);
    ~DiagramLine();

    QPoint offset() const { return QPoint((int)x(), (int)y()); }
    void drawShape(QPainter &painter);
    QPointArray areaPoints() const;
    int rtti() const { return RTTI; }
};

```

Abb. 91: Klassendefinition für das Canvas-Elemente Linie

Beide Klassendefinitionen haben also folgendes Ziel: Sie erben die Eigenschaft, ein Rechteck bzw. eine Linie zu zeichnen von den in Qt vordefinierten Klassen. Dabei erweitern sie ihre Klassen gegenüber der Basisklasse um die Eigenschaft, das umhüllende

Rechteck bzw. die Anfangs- und Endpunkte einer Linie zu ermitteln, geben einen RTTI-Wert zur Identifikation des Objekttyps zurück und ermöglichen beim Rechteck, darin einen Text hinzuzufügen.

Die Implementierung aller drei Klassen erfolgt in `cadview.cpp`. Dies ist möglich, da in C++ die Kompilierungseinheiten frei gewählt werden können. Ebenso hätte für jede Klasse eine eigene Kompilierungseinheit genutzt werden können.

Auf die Erläuterung des Konstruktors sowie der Implementierung der Aktionen von `CadView` wird an dieser Stelle verzichtet, da dies im vorherigen Kapitel bereits ausführlich dargestellt wurde und in diesem Beispiel entsprechend erfolgt.

Relevant für das Thema dieser Arbeit ist hier das Kontextmenü, welches unterschiedliche Menüpunkte (also Aktionen) darstellt, je nachdem ob ein Objekt oder der leere Arbeitsbereich geklickt wurde. Die Untersuchungen in Kapitel 2 haben ergeben, dass Kontextmenüs trotz ihrer Effizienz und situationskonformer Verständlichkeit kaum in die Programme des Bauwesens Einzug gefunden haben.

Die Programmierung eines Kontextmenüs ist elementar: Beim Mausklick muss eine Abfrage erfolgen, ob ein Element aktiviert wurde. Die dementsprechenden Aktionen werden dann in das Menü geladen (siehe Abb. 92).

Im Prototyp I war die eigene Entwicklung von Kontextmenüs nicht notwendig. Die dort verwendete Klasse `TextEdit`, die von `QTextEdit` abgeleitet worden war, verfügt bereits über ein in Qt vordefiniertes Kontextmenü mit den gebräuchlichen Aktionen der Textbearbeitung.

Im aktuellen Beispiel hingegen wird ein individuelles Kontextmenü erforderlich, welches vom Anwendungsentwickler auf die Elemente der Canvas abgestimmt werden muss.

Die Kontextmenüs sind in Bild 87 und 88 dargestellt. Bild 92 zeigt, wie über eine `if-else`-Abfrage ermittelt wird, ob ein Element angeklickt wurde, also aktiv ist (`activeItem`). Dementsprechend werden die Aktionen in das Menü geladen, die für diesen Kontext vorgesehen sind.

Ist kein Element aktiv (also Klick ins Leere), werden im `else`-Zweig die Aktionen zum Einfügen von Elementen eingefügt. Entsprechende Trennlinien (`insertSeparator`) strukturieren das Menü.

Nach Laden der entsprechenden Aktionen wird das Kontextmenü an der aktuellen Mausposition angezeigt.

```
//Methode contentsContextMenuEvent in Klasse CadView, formal
//ausgedrückt durch „:“
//Beim Aufruf wird event als Zeiger auf ein QContextMenuEvent übergeben
void CadView::contentsContextMenuEvent(QContextMenuEvent *event)
{
//Erstellt ein neues contextMenu vom Typ QPopupMenu
    QPopupMenu contextMenu(this);
//Ist ein Element ausgewählt (activeItem = true), also Klick auf ein Objekt,
//dann füge folgende Aktionen hinzu:
    if (activeItem) {
        cutAct->addTo(&contextMenu);
        copyAct->addTo(&contextMenu);
        deleteAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        bringToFrontAct->addTo(&contextMenu);
        sendToBackAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        propertiesAct->addTo(&contextMenu);
//Bei Klick in einen leeren Bereich:
    } else {
        pasteAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        addBoxAct->addTo(&contextMenu);
        addLineAct->addTo(&contextMenu);
    }
//Führe contextMenu an Mausposition aus:
    contextMenu.exec(event->globalPos());
}
```

Abb. 92: Quellcode zur Generierung des Kontextmenüs

Die Aktionen `addBoxAct` und `addLineAct` senden ein Signal an die Slots `addBox` sowie `addLine`. Die Definition der Signal-Slot-Verbindungen ist in diesem Beispiel nicht mehr dargestellt, da sie bereits im Prototyp I erörtert wurde und hier entsprechend erfolgt.

Die Slots `addBox` und `addLine` werden als Methoden in Abb. 93 definiert. Beide Methoden erzeugen in der Canvas über die Funktion `addItem` ein neues Element der Klasse `DiagramBox` bzw. `DiagramLine`, indem sie den jeweiligen Konstruktor aufrufen.

```
//Quellcode der paramaterlosen Funktion addBox der Klasse CadView
void CadView::addBox()
{
//übergibt DiagramBox an addItem, diese Funktion erzeugt dann das Rechteck
  addItem(new DiagramBox(canvas()));
}

void CadView::addLine()
{
  addItem(new DiagramLine(canvas()));
}
```

Abb. 93: Funktionen addBox() und addLine()

addItem bereitet das Einfügen des übergebenen Elements vor. Die Funktion ändert den Mauszeiger, und weist pendingItem das aktuelle Element zu, siehe Bild 94.

```
//Quellcode der Funktion addItem, beim Aufruf wird ein Q3CanvasItem übergeben
void CadView::addItem(Q3CanvasItem *item)
{
  //Evtl. vorhandenes Element „pendingItem“ löschen
  delete pendingItem;
  //Weist der Qt-Variablen pendingItem das aktuelle Element item zu
  pendingItem = item;
  //ruft setActiveItem(0) auf und setzt so pendingItem als aktiv
  setActiveItem(0);
  //ändert den Mauscursor zum Qt::crossCursor (plattformunabhängig das
  //Fadenkreuz zum Platzieren)
  setCursor(Qt::crossCursor);
}
```

Abb. 94: Funktion addItem in Klasse CadView

Das Element wird durch die Funktion addItem für die Platzierung auf der Canvas vorge-merkt, ist aber noch nicht sichtbar. Dies geschieht erst durch die Funktion contents-MousePressEvent (Abb. 95) bei folgendem Ereignis: Der Benutzer klickt in das An-wendungsfenster. Drückt der Benutzer die linke Maustaste, bestimmt er dadurch die Po-sition des Rechtecks, an dessen X- und Y-Koordinate der Ursprung des pendingItem platziert wird. Abschließend wird das neue Element angezeigt und der Mauszeiger wie-der in seine Normalform zurückgesetzt.

Alle anderen Maustastenergebnisse werden im else-Zweig als Versuch interpretiert, ein Element zu markieren oder eine Markierung aufzuheben, also den aktuellen Befehl, ein neues Element zu erzeugen, abzubereiten.


```
//Definition der Funktion contentsMouseEvent
//(erwartet ein QMouseEvent beim Aufruf)
void CadView::contentsMouseEvent(QMouseEvent *event)
{
    //Was soll geschehen, wenn linke Maustaste gedrückt wurde und ein pendingItem
    //vorhanden ist, also über addItem erzeugt und noch vorhanden?
    if (event->button() == Qt::LeftButton && pendingItem) {
        //pendingItem wird an die Mausposition verschoben
        pendingItem->move(event->pos().x(), event->pos().y());
        //pendingItem wird angezeigt
        showNewItem(pendingItem);
        //pendingItem wird zurückgesetzt auf null
        pendingItem = 0;
        //Cursor wird zurückgesetzt
        unsetCursor();

        //Was geschieht ansonsten, also wenn ein Mausklick geschieht aber kein
        //pendingItem vorhanden ist?
    } else {
        //Funktion collisions wird für aktuelle Mausposition aufgerufen und gibt eine
        //Liste aller Elemente zurück, die sich an der Mausposition befinden
        Q3CanvasItemList items = canvas()->collisions(event->pos());
        //ist die Liste leer, dann wird ein nicht vorhandenes Element als aktiv
        //gesetzt, also die Auswahl gelöscht
        if (items.empty())
            setActiveItem(0);
        //ansonsten wird erstes Element der Liste als aktives Element markiert
        else
            setActiveItem(*items.begin());
    }
    //aktuelle Mausposition wird auf lastPos gesichert
    lastPos = event->pos();
}
```

Abb. 95: Funktion contentsMouseEvent

Ein weiteres wesentliches GUI-Element von Grafikprogrammen ist das interaktive Verschieben von Elementen auf dem Bildschirm. Dies lässt sich mit Qt programmtechnisch lösen.

Beim Verschieben wird üblicherweise das Element mit der linken Maustaste ausgewählt und dann mit gedrückter linker Maustaste verschoben. Am Zielpunkt wird die Maustaste losgelassen.

Hierfür wird entsprechend Abb. 96 bei jedem durch die Mausbewegung hervorgerufenen Ereignis das Element um die Strecke, um die die Maus bewegt wurde, verschoben.

```
void CadView::contentsMouseMoveEvent(QMouseEvent *event)
{
    //Was geschieht, wenn linke Maustaste gedrückt ist und das Event
    //contentsMouseMoveEvent aktiv ist?
    if (event->state() & Qt::LeftButton) {

        //Ist ein activeItem vorhanden?
        if (activeItem) {
            //Dann wird das activeItem um die Differenz der Mausposition zu Beginn und
            //Ende des Mausklicks verschoben
            activeItem->moveBy(event->pos().x() - lastPos.x(),
                               event->pos().y() - lastPos.y());
            //lastPos des Mauszeigers wird neu gespeichert
            lastPos = event->pos();
            //Canvas muss aktualisiert werden
            canvas()->update();
        }
    }
}
```

Abb. 96: Elemente auf dem Bildschirm verschieben durch Funktion contentsMouseMoveEvent

Dies geschieht durch das fortlaufend ausgelöste Ereignis contentsMouseMoveEvent so lange, wie die Maus mit gedrückter linker Maustaste gezogen wird. event->state() & Qt::LeftButton ist das auslösende Ereigniskriterium. Sofern beide Bedingungen erfüllt sind und ein activeItem vorhanden ist, wird über einen Abgleich der X- und Y-Koordinaten das aktive Element verschoben und das Anwendungsfenster aktualisiert. Wann immer ein Canvas-Element verändert wird, muss die Canvas darüber informiert werden, dass sie neu gezeichnet werden muss.

Für das menschliche Auge erscheint dies als eine flüssige Bewegung, obwohl es sich um eine fortlaufende Serie von auslösenden Ereignissen mit vielen kleinen Verschiebungen im Pixelbereich handelt.

Weitere Funktionen, die für die Funktionalität der Anwendung formal notwendig, aber nicht in Zusammenhang mit dem Thema dieser Arbeit stehen, werden an dieser Stelle nicht weiter betrachtet und analog implementiert, so z.B. Funktionen zum Überlagern von Elementen („nach vorne“, „nach hinten“) oder zum Kopieren und Einfügen in und aus der Zwischenablage.

Für das Ziel dieser Arbeit sind speziell die Funktionen zur Erzeugung der Anfasserpunkte wichtig. Dies geschieht in der Implementierung der Klassen `DiagramBox` und `DiagramLine` ebenfalls in der Datei `cadview.cpp`.

Zunächst wird im `DiagramBox`-Konstruktor die vorläufige Größe des einen Stab repräsentierenden Rechtecks auf gut sichtbare 100x60 Pixel festgelegt, mit einem schwarzen Rand und einer weißen Füllung, siehe Bild 97. Als Standardbetextung wird „Stab“ auf Variable `str` zugewiesen. Wird der Konstruktor durch eine vom Benutzer ausgeführte Aktion aufgerufen, wird ein Rechteck mit diesen Eigenschaften auf der Canvas erstellt und gezeichnet.

```
//Konstruktor der Klasse DiagramBox
DiagramBox::DiagramBox(Q3Canvas *canvas)
    : Q3CanvasRectangle(canvas)
{
    //Größe des den Stab repräsentierenden Rechtecks wird gesetzt in Pixel
    setSize(100, 60);
    //Rand und Füllung
    setPen(QPen(Qt::black));
    setBrush(Qt::white);
    //Standardbetextung auf Variable str
    str = "Stab";
}
```

Abb. 97: Konstruktor für das Element Rechteck, das einen Stab repräsentiert

Da die Funktion `drawShape` normalerweise nur ein Rechteck zeichnet, hier aber weitere Attribute dargestellt werden sollen, ist die Funktion `drawShape` der Basisklasse in `DiagramBox` zu reimplementieren, d.h. zu überschreiben, siehe Bild 98. So soll die Klasse `CadView` zusätzlich zur Basisklasse (die nur ein Rechteck darstellt) einen Text im Rechteck darstellen können sowie kleine Rechtecke als Platzhalter für Anfasser an den Ecken des Rechtecks.

Zunächst wird die ursprüngliche Funktion `drawShape` der Basisklasse genutzt, um das Rechteck zu zeichnen. Danach wird mittig in das Rechteck der Text gezeichnet. Ist das Element aktiv, werden an den Ecken des Elements Rechtecke gezeichnet. Dazu wird die Funktion `drawActiveHandle` aufgerufen, der die Koordinatenpunkte übergeben werden, an denen so ein Anfasser gezeichnet werden soll. Hierzu wird z.B. der Rechteckpunkt oben rechts über die Eigenschaft `rect().topRight()` übergeben.

Die überschriebene Funktion `drawShape` zeichnet nun also nicht nur die Rechteckform, sondern fügt zudem Text hinzu und markiert das Objekt mit Anfassern, wenn es aktiv ist.

```
//Überschreiben der Funktion drawShape aus Basisklasse
void DiagramBox::drawShape(QPainter &painter)
{
//nutzt drawShape der Basisklasse Q3CanvasRectangle um das Rechteck
//zu erzeugen
    Q3CanvasRectangle::drawShape(painter);
//Zeichnet Text in das Rechteck mit zentrierter Ausrichtung
    painter.drawText(rect(), Qt::AlignCenter, text());
//Ist das Element aktiv, zeichnet die Funktion drawActiveHandle
//Anfasser an das Element
    if (isActive()) {
        drawActiveHandle(painter, rect().topLeft());
        drawActiveHandle(painter, rect().topRight());
        drawActiveHandle(painter, rect().bottomLeft());
        drawActiveHandle(painter, rect().bottomRight());
    }
}
```

Abb. 98: Überschreiben der Funktion `drawShape()`

Die Funktion `drawActiveHandle` aus Bild 99 erzeugt ein kleines Quadrat. Dazu werden der Funktion Koordinatenpunkte übergeben (siehe Bild 98), an denen ein Rechteck in entsprechenden Abmessungen gezeichnet wird.

```
//Deklaration einer Konstanten Margin mit Wert 2
const int Margin = 2;
//Funktion drawActiveHandle
void drawActiveHandle(QPainter &painter, const QPoint &center)
{
//Setzen der Umrandung und der Farbe
    painter.setPen(Qt::black);
    painter.setBrush(Qt::gray);
//Platzieren eines Rechtecks: X-Koordinate, Y-Koordinate, Breite, Höhe
    painter.drawRect(center.x() - Margin, center.y() - Margin,
        2 * Margin + 1, 2 * Margin + 1);
}
```

Abb. 99: Erzeugen der Anfasser

Für die Implementierung von `DiagramLine` wird entsprechend verfahren.

Die so erzeugten Anfasserpunkte markieren lediglich das Objekt. Dieses Objekt kann dadurch nicht direkt unmittelbar in der Größe verändert werden. Die Qt-Bibliothek verfügt

über keine Funktion, die diese grundsätzliche Forderung von [Chan02] umsetzt, und bietet somit kein für Softwareunternehmen der Bauindustrie wirtschaftliches Verfahren. Dieser Mangel wurde auch von Trolltech erkannt und soll zukünftig beseitigt werden, da die Marktbedeutung erkannt wurde.

Erstrebenswert bleibt somit das Ziel, mit den vorgefertigten Mitteln einer Programmiersprache bzw. Programmierumgebung diese elementaren Funktionen einer modernen Benutzungsoberfläche programmtechnisch zu lösen. Nur so kann es für kleine Softwareunternehmen wirtschaftlich gelingen, effiziente und verständliche Benutzungsoberflächen zu gestalten.

Der nächste Software-Prototyp wird die mustergültige Lösung in .NET vorstellen und modellhaft implementieren.

Die bisherigen Programmbeispiele zeigen immerhin die Möglichkeiten, mit Qt klassische Elemente einer Benutzungsoberfläche zu gestalten.

Diese Software-Prototypen geben einen Einblick in die komplexe Materie der Grafikprogrammierung und zeigen auf, wie umfangreich die Entwicklung eines CAD-Systems ist, das den Anforderungen aus Kapitel 3 entspricht. Zudem wird gezeigt, wie die in dieser Arbeit analysierten, marktführenden CAD-Systeme BOCAD-3D und Tekla Structures zu ertüchtigen sind, damit auch sie zukünftig den ermittelten Weltmarkt-Anforderungen an Effizienz und Verständlichkeit entsprechen.

5.5 Software-Prototyp IV: Direkte, interaktive Manipulation mittels Anfassen

Mit Markteinführung von Windows Vista Anfang 2007 präsentierte Microsoft mit der Windows Presentation Foundation (WPF) eine neue Programmierschnittstelle für Grafik, Video und Audio. Diese wird die veraltete Win32-API ablösen. WPF nutzt das .NET Framework 3.0 und ist somit nicht nur für Rechner mit Windows Vista verfügbar, sondern auch auf Windows Server 2003 und Windows XP mit Servicepack 2 lauffähig. Durch das bereits vorgestellte MONO-Projekt, eine Open-Source Initiative für eine .NET-kompatible Laufzeitumgebung, sind WPF-Anwendungen auch plattformunabhängig.

WPF-Anwendungen können als klassische Desktop-Programme ausgeführt werden. Besonders interessant für Kooperationen von Ingenieuren in weltweiten Netzen ist aber die Eigenschaft, ohne Quellcodeanpassung als verteilte Anwendung z.B. im Internet über marktgängige Browser genutzt zu werden.

Dabei verfolgt Microsoft mit der WPF einen grundsätzlich neuen Ansatz bei der Anwendungsprogrammierung, in dem Anwendungslogik und Oberflächengestaltung strikt getrennt werden. Petzold formuliert in [Petz06] entsprechend: „Anwendung = Code + Markup“.

Dabei bezeichnet Markup die Gestaltung der Oberfläche in der mit der WPF neu geschaffenen Markupsprache XAML (eXtensible Application Markup Language). Laut [WIKI-Markup] dienen Markup Sprachen zur Beschreibung von Daten und deren Verarbeitung mittels Auszeichnungen. Die wohl bekannteste Markupsprache ist sicherlich HTML (Hyper Text Markup Language). Die Herkunft dieser Bezeichnung kommt aus der Druckindustrie, in der Auszeichnungen (Markups) im Text Hinweise an den Setzer gaben.

Dem Oberflächendesigner steht ein Gestaltungsprogramm zur Verfügung, das die Benutzungsoberfläche in XAML liefert. Mit Stand März 2007 ist dafür Microsoft Expression Blend als Beta-Version verfügbar, wie auch in dieser Arbeit eingesetzt.

Für die Programmierung der Anwendungslogik steht den Entwicklern jede von .NET unterstützte Programmiersprache frei. In dieser Arbeit wird C# als moderne Hochsprache eingesetzt. Als Entwicklungsumgebung dient Microsoft VisualStudio C# Express Edition, welches kostenlos verfügbar ist.

Im Software-Prototypen IV wird nun experimentell nachgewiesen, was mit der Qt-Bibliothek nicht möglich war: Anfasserelemente, am Objekt platziert, die dieses direkt manipulieren. .NET bietet dazu ein mustergültiges Vorgehen, welches die Implementierung in Anwendungen des Bauwesens auch für kleine Softwarehäuser wirtschaftlich möglich macht.

Bild 100 zeigt das Prinzip. Jedem Element auf der Benutzungsoberfläche (adorned Element) kann ein Schmuckelement (Adorner; von „to adorn“, schmücken) auf einem Adornerlayer zugewiesen werden. Als Adorner können sämtliche Elemente einer Benut-

zungsoberfläche genutzt werden, z.B. Texte, Symbole oder Schaltflächen. Für Anfasser, die das direkte Manipulieren von Objekten ermöglichen, verfügt die .NET-Bibliothek über „Thumbs“ mit vordefinierten Mausereignissen.

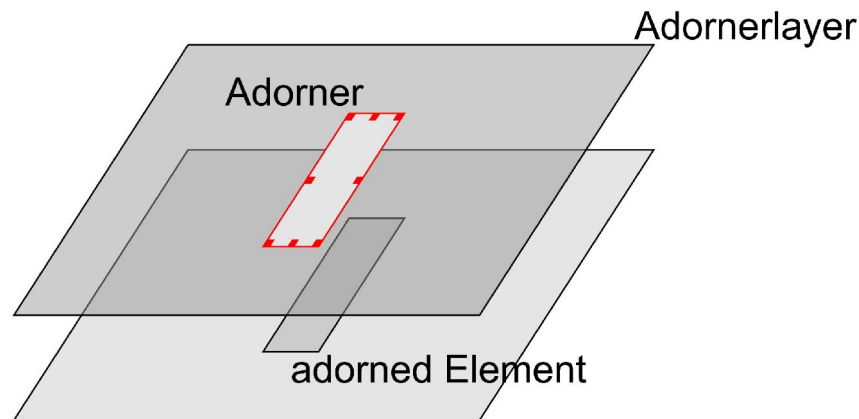


Abb. 100: Adorner-Prinzip in .NET

Diese Werkzeuge werden nun genutzt, um im bereits vorgestellten, vereinfachten CAD-System des Prototyps III direkte Manipulation über Anfasser exemplarisch zu verwirklichen.

Dazu wird ein Stabelement mit Anfassern dargestellt, welches der Benutzer mit der Maus verschieben und in seinen Abmessungen verändern kann.

Der Mauszeiger ändert sich beim Verschieben entsprechend zu einem Cursorkreuz (⊕). Klickt der Benutzer den Stab an, erscheinen Anfasser an seinen Ecken sowie in den Mittelpunkten der Außenkanten. Fährt die Maus über einen solchen Anfasser, verändert sich der Mauscursor und signalisiert dem Benutzer, dass er mit diesem Anfasser die Abmessungen des Stabes verändern kann (Bild 101).

Das Ziehen des Anfassers in den Eckpunkten verändert die Länge und Breite des Stabes proportional. Wird ein Anfasser im Mittelpunkt der Stirnkante gezogen, ändert sich nur die Länge des Stabes, beim Ziehen des Anfassers in der Mitte der Seitenkanten ändert sich nur die Breite des Stabes. So können die Abmessungen sowie der Profiltyp des Stabes direkt manipuliert werden.

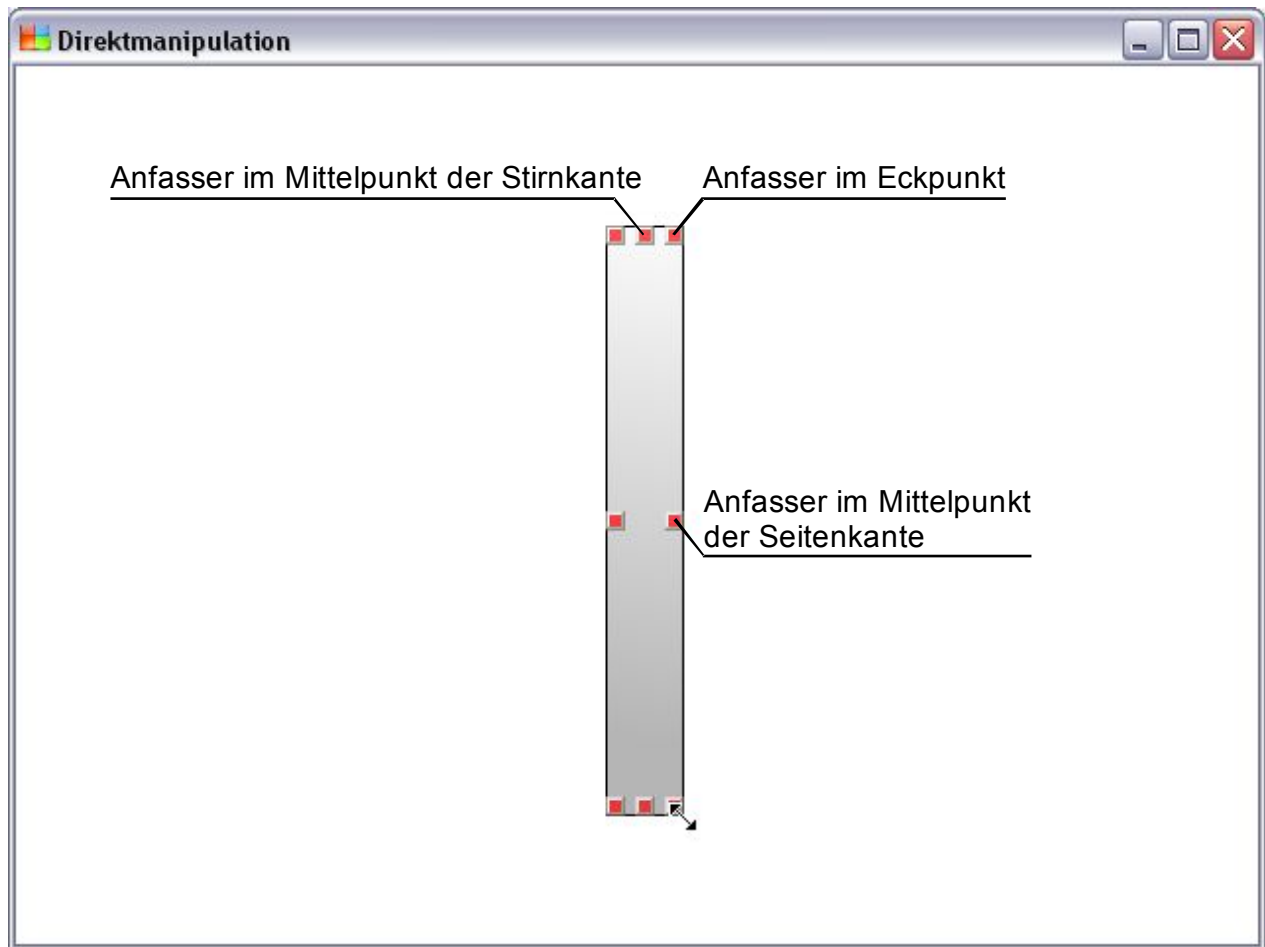


Abb. 101: Software-Prototyp IV, modellhaftes CAD-System, Objekt mit Anfassern zur direkten Manipulation

Zunächst wird die Benutzungsoberfläche der Anwendung in Expression Blend entworfen, siehe Bild 102.

Über die Werkzeugleiste (Bereich 1 in Bild 102) lassen sich neue Elemente hinzufügen, z.B. Schaltflächen oder Grafikobjekte.

Bereich 2 in Bild 102 listet alle vorhandenen Objekte auf. Für dieses Beispiel ist lediglich ein Anwendungsfenster („FirstWindow“) mit einem „LayoutRoot“ notwendig. Dies wird automatisch mit jedem neuen Projekt angelegt.

Das Element „FirstObject“ ist das in Layoutbereich (3) sichtbare Rechteck. In diesem Bereich ist neben der in Bild 102 gezeigten Designansicht auch die Ansicht des erzeugten XAML-Codes möglich, der von Expression Blend automatisch erzeugt wird.

Bereich 4 ist das Eigenschaftsfenster des gerade aktiven Objektes, in diesem Fall des Rechtecks. Dieser Bereich ist kontextabhängig. Hier werden auch die Ereignisse, mit denen ein Element verknüpft werden soll, definiert.

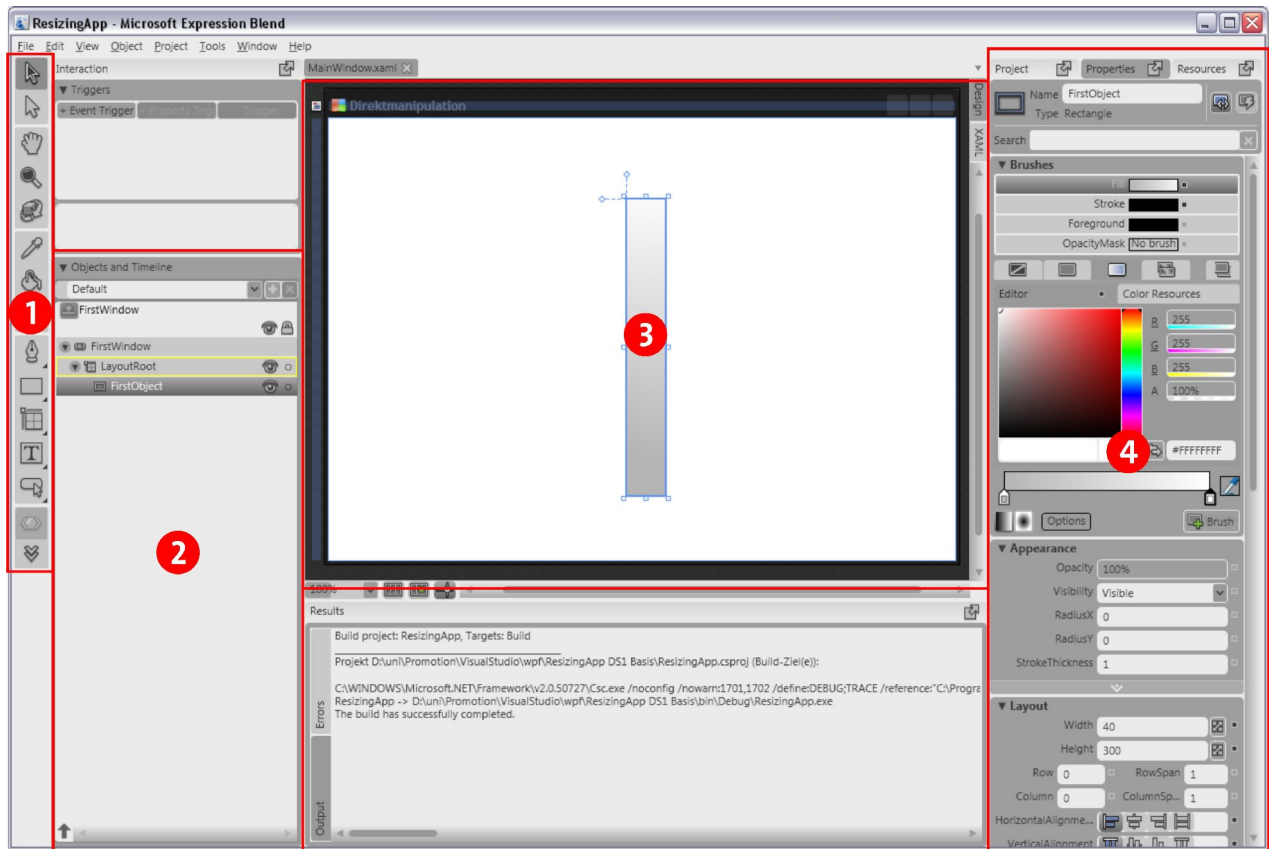


Abb. 102: Microsoft Expression Blend, Gestaltung der Benutzungsoberfläche

Der von Expression Blend erzeugte XAML-Code wird in der Datei `MainWindow.xaml` gespeichert. Er ist für diese Arbeit nicht von Interesse und wird daher nicht weiter vorgestellt.

Nach dem Gestaltungsentwurf der Oberfläche werden die Mausereignisse definiert, auf die das Programm entsprechend reagieren soll.

Dazu bietet Expression Blend eine Übersicht über alle „Events“ (Ereignisse) an, auf die in Bereich 4 umgeschaltet werden kann, siehe Bild 103. Ereignisse können jedem Element zugewiesen werden.

Bild 103 zeigt die vordefinierten Ereignisse, die ausgelöst werden können. Diese müssen mit entsprechenden Funktionen verbunden werden, die in einer „Code behind“-Datei

implementiert werden. Damit ist der Anwendungscode gemeint, der „hinter“ der Oberfläche liegt und dazu dient, die Benutzereingaben zu verarbeiten. Er wird nicht in XAML geschrieben sondern in der gewählten Programmiersprache, hier also in C#.

Für diese Anwendung werden folgende Ereignisse genutzt, also während des Programm- laufs überwacht:

- Drücken der linken Maustaste (`MouseDown`) ruft Funktion `CaptureLeftButton` auf
- Loslassen der linken Maustaste (`MouseLeftButtonUp`) ruft Funktion `StopCaptureLeftButton` auf
- Bewegen der Maus (`MouseMove`) ruft Funktion `DragObject` auf
- Drücken einer Tastaturtaste (`KeyDown`) ruft Funktion `CaptureKey` auf
- Loslassen einer Tastaturtaste (`KeyUp`) ruft Funktion `StopCaptureKey` auf.

Die „Code behind“-Datei wird nach WPF-Konvention `MainWindow.xaml.cs` benannt. In dieser werden die zuvor mit den Ereignissen verknüpften Funktionen implementiert.

Bild 104 zeigt das automatisch erzeugte Grundgerüst einer „Code behind“-Datei. Zunächst müssen „Namespaces“ (Namensräume) eingebunden werden. Namensräume sind vergleichbar zu Ortsvorwahlen im Telefonnetz. So können Variablen (Telefonnummern) mehrmals vergeben werden, wenn sie sich in unterschiedlichen Namensräumen (Ortsvorwahlen) befinden. Um Variablen oder

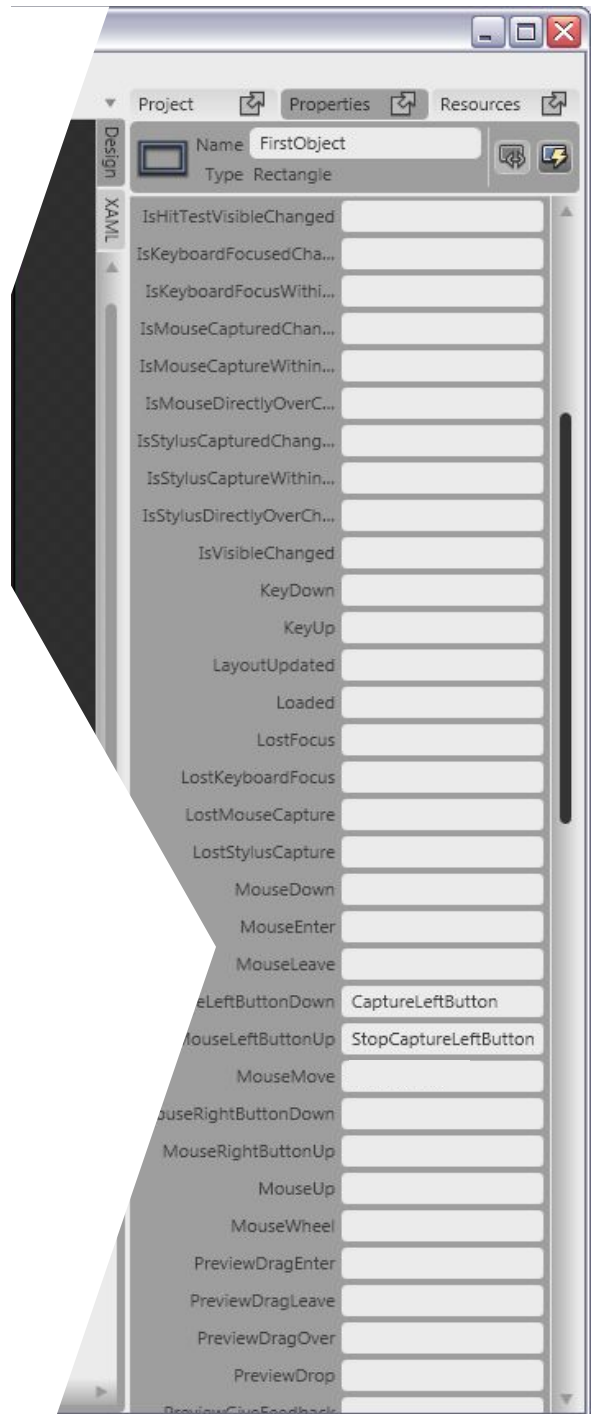


Abb. 103: Übersicht über Ereignisse, die ausgelöst werden können

Methoden aus einem fremden Namensraum nutzen zu können, muss der entsprechende Namensraum über `using` eingebunden werden. Der Grundgedanke hinter diesen Namensräumen ist mit benannten Common-Blöcken der klassischen Programmiersprache Fortran vergleichbar.

Für diese Anwendung wird ein neuer Namensraum `ResizingApp` generiert.

```
//Einbinden der Namespaces
using System;
using System.IO;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Navigation;

//Erstellen eines eigenen Namespaces "ResizingApp"
namespace ResizingApp
{
    //Definition einer partiellen Klasse MainWindow
    public partial class MainWindow
    {
        //Konstruktor der Klasse, initialisiert das Objekt wenn der Konstruktor
        //aufgerufen wird
        public MainWindow()
        {
            this.InitializeComponent();
        }

        //Eigener Code der Klasse, also Methoden und Attribute
    }
}
```

Abb. 104: Grundgerüst einer "Code behind"-Datei

Es wird eine neue Klasse `MainWindow` erzeugt. Das Schlüsselwort `partial class` gibt an, dass sich die Klassendefinition über mehrere Quelldateien aufteilt. So wird ein Teil der Klasse in der Code-Datei definiert, ein anderer Teil durch die XAML-Datei. Der Konstruktor initialisiert das Objekt. Das C#-Schlüsselwort im Konstruktor führt die Anwendung schlussendlich aus.

Alle folgenden Funktionen werden als Membermethoden in der Klasse `MainWindow` implementiert, d.h. an Stelle des Kommentars „Eigener Code“ aus Bild 104.

Zunächst wird der Adornerlayer geladen. Dies geschieht in der Funktion `GetAdornerLayer`, die über das Ereignis `Loaded` in `MainWindow` direkt nach dem Start der Anwendung ausgeführt wird. Bild 105 zeigt die Funktion. Über vordefinierte Schlüsselwörter wird ein neuer `AdornerLayer` aus dem Objekt `LayoutRoot`, dem Anwendungsfenster, erzeugt.

```
//private-Funktion GetAdornerLayer
private void GetAdornerLayer(object sender, System.Windows.RoutedEventArgs e)
{
//Hinzufügen eines AdornerLayer auf der Variablen adornerLayer
    adornerLayer = AdornerLayer.GetAdornerLayer(LayoutRoot);
}
```

Abb. 105: Hinzufügen des Adornerlayer

Nachdem der Adornerlayer initialisiert aber noch ohne Inhalt ist, müssen Maus- und Tastaturereignisse implementiert werden.

Dazu werden zunächst einige Funktionen implementiert, die noch keine Rückmeldung an den Benutzer geben. Sie dienen vielmehr „im Hintergrund“ dazu, Werte zu ermitteln (z.B. die Mausposition) oder Variablen zu setzen (z.B. ob die linke Maustaste gedrückt ist).

Wird die linke Maustaste gedrückt, sendet das aktive Objekt, hier das Rechteckelement, das Ereignis `MouseLeftButtonDown` und führt so die Funktion `CaptureLeftButton` aus.

Bild 106 zeigt, dass zunächst drei Variablen deklariert und initialisiert werden. Ihre Werte werden innerhalb der Funktion verändert. Beim Aufruf der Funktion wird das Objekt, welches das Ereignis ausgelöst hat, als `sender` an die Funktion übergeben ebenso wie die Argumente des Ereignisses.

```
//Variablendeklaration und -initialisierung
public static bool LeftMouseDown = false;
public static double MouseLeft = 0;
public static double MouseTop = 0;

private void CaptureLeftButton(object sender,
System.Windows.Input.MouseButtonEventArgs e)
{
//sender, das Objekt, welches die Funktion aufruft, wird auf uie als
//UIElement gecastet, d.h. das Rechteck wird referenziert
    UIElement uie = sender as UIElement;
//Die Mausposition innerhalb des Elements wird ermittelt
    MouseLeft = e.GetPosition(uie).X;
    MouseTop = e.GetPosition(uie).Y;
//Flag LeftMouseDown wird gesetzt
    LeftMouseDown = true;
}
```

Abb. 106: Funktion CaptureLeftButton ermittelt die Mausposition und setzt Flag

sender als Variable vom Typ UIElement wird nun auf uie vom Typ UIElement gecastet. Typcasting bedeutet die Umwandlung eines Wertes von einem Datentyp in einen anderen. In diesem Fall wird sender, welches aus dem Funktionsaufruf vom Typ object ist, in ein UIElement gewandelt. Danach wird das Ergebnis auf uie, ebenfalls vom Typ UIElement, gespeichert.

Auf MouseLeft bzw. MouseTop wird die Position des Mausursors innerhalb des Objektes uie gespeichert. Dazu werden die übergebenen Ereignisargumente genutzt. Die Mausposition wird später beim Verschieben des Objektes wieder benötigt.

Die Variable LeftMouseDown wird als aktiv gesetzt, d.h. sie wird als Flag genutzt, dass die linke Maustaste aktuell gedrückt ist.

Entgegen der Vorstellung, dass der Klick mit der linken Maustaste ein Objekt als markiert anzeigt, ist es vielmehr das Loslassen der linken Maustaste, das eben diese Aktion ausführt. Denn das Drücken der linken Maustaste kann zunächst zwei Resultate beabsichtigen:

Der Benutzer möchte das Objekt markieren, d.h. er wird die linke Maustaste drücken und wieder loslassen, ohne die Maus zu bewegen. Oder der Benutzer möchte das Objekt verschieben. Dazu wird er die linke Maustaste niederdrücken, gedrückt halten und danach die Maus mit gedrückter Maustaste bewegen. Dann soll auch das Objekt entsprechend verschoben werden.

Dies wird in den Funktionen `StopCaptureLeftButton` und `DragObject` unterschieden. `StopCaptureLeftButton` wird aufgerufen, wenn der Benutzer die linke Maustaste loslässt. Sie muss also überprüfen, ob der Benutzer

- ein Element erstmalig anklickt, so dass der Adorner hinzugefügt und das Element als aktiv gesetzt wird
- ein aktives Element erneut angeklickt hat, so dass Adorner zu entfernen und das Element als nicht aktiv zu setzen ist.

Dies darf allerdings nur geschehen, wenn der Benutzer nicht gleichzeitig auch die Maus bewegt.

Bewegt der Benutzer die Maus, ruft das Ereignis `MouseMove` die Funktion `DragObject` auf. Diese überprüft

- ob die linke Maustaste gedrückt ist, denn nur dann soll eine Reaktion erfolgen,
- ob der Adorner aktuell angezeigt wird, um diesen jetzt beim Verschieben des Objekts zu entfernen,

und verschiebt dann das Objekt über den Bildschirm, der Mausbewegung folgend.

Diese Funktionen werden nun im Detail vorgestellt. Bild 107 zeigt `StopCaptureLeftButton`. Zunächst wird abgefragt, ob das Flag `HasBeenDragged`, welches in `DragObject` aktiv gesetzt wird, aktuell nicht aktiviert ist. So erkennt das Programm, dass der Benutzer die Maus aktuell nicht bewegt.

Wurde durch den Mausklick ein Element ausgewählt, ist zu überprüfen, ob das Element bereits aktiv war, um den Adorner zu entfernen, oder ob das Element durch diesen Mausklick erst aktiviert wurde, um den Adorner hinzuzufügen.

Dazu wird abgefragt, ob das aktuell ausgewählte Element (`uie`) identisch mit dem `activeElement` ist und ob aktuell `activeAdorner` vorhanden ist. Nur dann wird der `adornerLayer` entfernt.

```

//Fügt einen Adorner an ein vorher nicht aktives Element bzw. Entfernt
//diesen bei erneutem Mausklick am aktiven Element
private void StopCaptureLeftButton(object sender,
System.Windows.Input.MouseButtonEventArgs e)
{
//Ist das Flag HasBeenDragged nicht gesetzt, wird das Objekt also
//aktuell NICHT bewegt?
    if (!HasBeenDragged)
    {
        UIElement uie = sender as UIElement;
//Ist ein Element ausgewählt?
        if (uie != null)
        {
//Ist aktuell uie (das Objekt, welches die Funktion aufruft) identisch
//mit dem aktiven Element und ein activeAdorner vorhanden?
            if (uie == activeElement && activeAdorner != null)
            {
//dann entferne den adornerLayer und setze activeElement
//und activeAdorner zurück
                adornerLayer.Remove(activeAdorner);
                activeElement = null;
                activeAdorner = null;
            }
//wenn kein Objekt gewählt, also uie nicht activeElement, dann
//füge neuen ResizingAdorner auf adornerLayer hinzu und setze uie als
//activeElement
            else
            {
                activeAdorner = new ResizingAdorner(uie);
                adornerLayer.Add(activeAdorner);
                activeElement = uie;
            }
        }
    }
//Setze LeftMouseDown und HasBeenDragged zurück
    LeftMouseDown = false;
    HasBeenDragged = false;
}

```

Abb. 107: Funktion StopCaptureLeftButton

Ansonsten wird ein neuer Adorner erzeugt. Dazu wird dieser als neues Objekt aus der Klasse `ResizingAdorner` instanziiert. Diese Klasse wird später in der Datei `resizingadorner.cs` definiert.

Am Ende der Funktion werden die Flags `LeftMouseDown` und `HasBeenDragged` zurückgesetzt, da die linke Maustaste nun nicht mehr gedrückt wird und die „Drag“-Aktion beendet ist. Geschieht dies nicht, würde die Anwendung weiterhin davon ausgehen, dass die linke Maustaste gedrückt wird.

Klickt der Benutzer hingegen mit der linken Maustaste ein Element an, macht es dadurch aktiv, und bewegt die Maus bei weiterhin gedrückter linker Maustaste, soll das ausgewählte Element verschoben werden. Dies geschieht über die Funktion `DragObject`, aufgerufen durch `MouseMove` in `MainWindow`.

```
//Verschiebt Objekt
private void DragObject(object sender, System.Windows.Input.MouseEventArgs e)
{
    //Ist linke Maustaste gedrückt?
    if (LeftMouseIsDown)
    {
        //Ist ein Adorner aktuell vorhanden?
        if (activeAdorner != null)
        {
            //dann entferne den Adorner
            adornerLayer.Remove(activeAdorner);
            activeAdorner = null;
        }
        //Flag HasBeenDragged wird gesetzt
        HasBeenDragged = true;
        //Mausposition im Fenster
        double posXInWindow = e.GetPosition(LayoutRoot).X;
        double posYInWindow = e.GetPosition(LayoutRoot).Y;
        //Element wird verschoben durch Verändern des linken und oberen Abstandes
        //zum Fensterrand
        //Änderung wird ermittelt durch Differenz von Mausposition im Objekt zu
        //Mausposition im Fenster
        Thickness dicke = new Thickness(posXInWindow - MouseLeft, posYInWindow -
        MouseTop, 0, 0);
        FirstObject.Margin = dicke;
    }
}
```

Abb. 108: Funktion `DragObject`

Zunächst wird abgefragt, ob die linke Maustaste gedrückt ist. Dazu wird das Flag `LeftMouseIsDown` genutzt. Ist es aktiv, ist die Maustaste gedrückt, da es erst beim Loslassen der Maustaste in `StopCaptureLeftMouse` zurückgesetzt wird.

Es entspricht gewohntem Systemverhalten, dass Anfasser beim Verschieben eines Objektes nicht angezeigt werden. Dazu wird `activeAdorner` überprüft. Ist das Objekt vorhanden, wird der Adorner aktuell angezeigt und daher entfernt.

Das Flag `HasBeenDragged` wird gesetzt, um in `StopCaptureLeftButton` zu verhindern, dass ein Loslassen der Maustaste beim Bewegen der Maus die Adorner anzeigt (siehe Bild 107).

Nachdem in `CaptureLeftButton` bereits die Mausposition im Objekt (hier Rechteck) als `MouseLeft` und `MouseTop` ermittelt wurde, wird nun zusätzlich die Mausposition im Fenster als `posXInWindow` und `posYInWindow` ermittelt.

Bild 109 zeigt, dass in XAML Elemente über den Abstand zur linken oberen Fensterecke als Nullpunkt platziert werden. X- und Y-Abstand zur linken, oberen Fensterecke werden `Margin Top` bzw. `Margin Left` genannt.

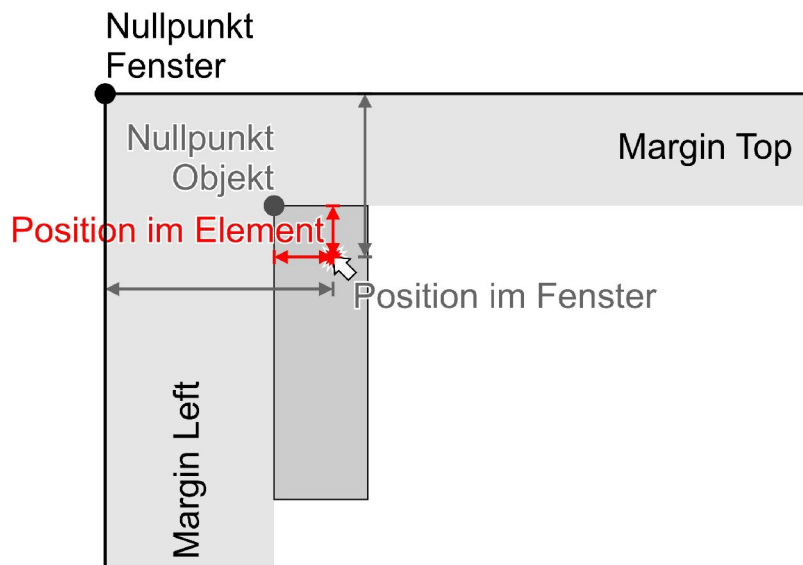


Abb. 109: Verschieben eines Elementes bezogen auf die linke obere Fensterecke als Nullpunkt

Um das ausgewählte Element somit verschieben zu können, müssen die X- und Y-Koordinaten der Stelle, an der mit der Maus geklickt wurde, relativ zum Objekt-Nullpunkt umgerechnet werden. Dazu wird die Differenz zwischen der Position des Mausklicks im Fenster und der Position im Element ermittelt.

Durch permanentes, wiederholtes Durchlaufen der Funktion `DragObject` solange, wie der Benutzer die Maus bewegt, berechnet sich somit in einer Schleife ständig die Mausposition im Fenster neu. Nach Subtraktion der (feststehenden, da nur beim Niederdrücken der Maustaste in `CaptureLeftButton` einmalig ermittelten) Position im Objekt kann so der Randabstand des Objekt-Nullpunkts vom Fenster-Nullpunkt fortlaufend berechnet und verändert werden. Diese ständige der Mausposition folgende Neupositionierung erscheint dem Benutzer als eine kontinuierliche, flüssige Bewegung seit Rechen- und Grafikhardware hierfür genügend leistungsfähig sind.

Objekte können ohne Verschieben angeklickt werden. Ein so gewähltes Objekt wird aktives Objekt, was auf dem Bildschirm durch Adorner am Objekt angezeigt wird. Dazu muss also auf dem Adornerlayer ein Adorner für das Objekt hinzugefügt werden. Dieser `activeAdorner` wird als Objekt der Klasse `ResizingAdorner` instanziiert. Diese Klasse wird in `resizingadorner.cs` implementiert, siehe Bild 110.

```
public class ResizingAdorner : Adorner
{
    // Resizing adorner nutzen Thumbs für visuelle Darstellung.
    // Die Thumbs haben vordefinierte Mausereignisse.
    Thumb topLeft, topRight, bottomLeft, bottomRight, Top, Bottom,
    Left, Right;
}
```

Abb. 110: `ResizingAdorner` wird abgeleitet von `Adorner`

Die Klasse `ResizingAdorner` platziert 8 Anfasser an das Element, mit denen die Größe des „geschmückten“ Elements verändert werden kann.

Dazu bietet die .NET Bibliothek die Klasse `Thumb` als Grundlage für Anfasser. Diese verfügt über vordefinierte Mausereignisse, die das Implementieren der Anfasser und ihrer Funktionalität vereinfachen. Sie müssen somit nicht von Softwareentwicklern selbst entwickelt werden, was für Softwareunternehmen des Bauwesens eine unlösbare Herausforderung darstellt. Sie müssten nämlich dann selbst entsprechende Bibliotheken und Werkzeuge wie Qt oder Microsoft mit .NET entwickeln.

So ist genau diese Klasse „Thumb“ *conditio sine qua non* für die Schaffung des in dieser Arbeit geforderten, für kleine Softwareunternehmen wirtschaftlichen programmtechnischen Ansatzes für textfreie Benutzungsoberflächen mit direkter Manipulation der Objekte auf dem Bildschirm.

Die Klasse `ResizingAdorner`, die nun folgend vorgestellt wird, bietet so die Basis für effiziente Benutzungsoberflächen im Bauwesen. Sie kann nahezu unverändert in jegliche Bausoftware übernommen werden und prägt so eine neue Generation effizienter Benutzungsoberflächen.

Bild 110 zeigt, dass `ResizingAdorner` von der Basisklasse `Adorner` abgeleitet und weiter ergänzt wird, indem 8 Thumbs platziert und mit entsprechenden Events verknüpft werden.

Die Variablen `topLeft`, `topRight` etc. bezeichnen die späteren Anfasserelemente und werden als Objekte der Klasse `Thumb` definiert.

```
public ResizingAdorner(UIElement adornedElement) : base(adornedElement)
{
    //visualChildren als neues Objekt der Klasse VisualCollection
    visualChildren = new VisualCollection(this);

    //Hilfsmethode erzeugt die Thumbs
    //mit benutzerdefinierten Cursorformen
    BuildAdornerCorner(ref topLeft, Cursors.SizeNWSE);
    BuildAdornerCorner(ref topRight, Cursors.SizeNESW);
    BuildAdornerCorner(ref bottomLeft, Cursors.SizeNESW);
    (...)
    //Fügt der Eigenschaft DragDelta der Thumbs ein weiteres Event hinzu
    //z.B. HandleBottom
    (...)
    Bottom.DragDelta += new DragDeltaEventHandler(HandleBottom);
    Left.DragDelta += new DragDeltaEventHandler(HandleLeft);
    Right.DragDelta += new DragDeltaEventHandler(HandleRight);
}
```

Abb. 111: Konstruktor der Klasse `ResizingAdorner`

Der Konstruktor der Klasse `ResizingAdorner` (Bild 111) erzeugt `Thumbs` über die Funktion `BuildAdornerCorner`. Danach wird dem in `Thumb` vordefinierten Ereignis `DragDelta` (also das Erkennen einer Verschiebung des Elements) ein selbstdefiniertes Event hinzugefügt, damit jeder Anfasser entsprechend seiner Position am Objekt eine „intelligente“ spezifische Objektveränderung bewirken kann.

Die Funktion `ArrangeOverride` der Klasse `Adorner` muss folglich überschrieben werden, um die `Thumbs` zu platzieren. Bei der Instanzierung eines Objektes der Klasse `ResizingAdorner` wird diese Funktion aufgerufen und ordnet die `Thumbs` entsprechend an, siehe Bild 112.

Dazu wird zunächst die Höhe und Breite des rechteckigen `adornedElement` als `adornerWidth` und `adornerHeight` ermittelt. Diese Größenangaben sind inkl. Randabstandes zum Fenster-Nullpunkt, der daher subtrahiert wird. Dazu wird das `adornedElement` als `rec` gecastet. So kann der Randabstand links und oben ausgelesen und von `adornerWidth` und `adornerHeight` subtrahiert werden. Das Ergebnis ist die Größe des `adornedElement` ohne Rand.

Objekte mit anderer als rechteckiger Form erfordern dabei einen entsprechend modifizierten Ansatz. Denn das in diesem beispielhaften Prototypen verwendete Rechteck zur Darstellung eines Stabes stellt eine Vereinfachung dar. In CAD-Systemen finden vielfach auch Polygonzüge Anwendung. Bei der Markierung soll dann nicht das umschließende Rechteck ermittelt werden, sondern an jedem Anfangs- und Endpunkt einer Polygonlinie sollen sich Anfasser zum Verschieben befinden.

Dieses Beispiel zeigt so vereinfacht und somit besonders lehrhaft die prinzipielle Anwendung des Thumb-Prinzips, dass entsprechend auf andere Objekte anwendbar ist.

```
//Überschreiben der Funktion ArrangeOverride
protected override Size ArrangeOverride(Size finalSize)
{
//adornedWidth und adornedHeight sind Breite und Höhe des adornedElement
//inkl Rand
    double adornerWidth = AdornedElement.DesiredSize.Width;
    double adornerHeight = AdornedElement.DesiredSize.Height;
//Hälfte der Thumbsgröße zum späteren Platzieren innerhalb des Elements
    double adornerSize = 5;
//AdornedElement als rec verfügbar
    Rectangle rec = AdornedElement as Rectangle;
//Berechnen der desiredWidth ohne linken und oberen Rand
    adornerWidth = adornerWidth - rec.Margin.Left;
    adornerHeight = adornerHeight - rec.Margin.Top;
//Platzieren des linken, oberen Thumbs mithilfe eines "Hilfsrechtecks" an den
//angegeben Koordinaten
    topLeft.Arrange(new Rect(-adornerWidth / 2 + adornerSize,
-adornerHeight / 2 + adornerSize, adornerWidth, adornerHeight));
//für die anderen Thumbs entsprechend
    topRight.Arrange(new Rect(adornerWidth / 2 - adornerSize,
-adornerHeight / 2 + adornerSize, adornerWidth, adornerHeight));
    bottomLeft.Arrange(new Rect(-adornerWidth / 2 + adornerSize, adornerHeight
/ 2 - adornerSize, adornerWidth, adornerHeight));
    (...)
}
```

Abb. 112: Anordnen der Thumbs

Nach Ermittlung der Objektgröße können nun die einzelnen Thumbs platziert werden. Dies geschieht über die Funktion `arrange`. Referenzpunkt ist dabei das Zentrum des Adorners. Es wird ein neues Rechteck erzeugt, welches z.B. für den linken oberen Anfasser um die halbe Adornerbreite nach links und die halbe Adornerhöhe nach oben verschoben wird. Breite und Höhe des Rechtecks sind identisch mit den Abmessungen des Adorners, aber nicht weiter von Interesse, da das Rechteck unsichtbar bleibt, d.h. keine Füllung oder Rand erhält. Das so erzeugte Rechteck dient lediglich für die Platzierung

des bereits definierten Thumb, welches mittig auf die linke obere Ecke des Rechtecks platziert wird.

Bild 113 zeigt die Funktion `BuildAdornerCorner`, die ein einzelnes Thumb erzeugt. Die Funktion ist allgemein gehalten und wird entsprechend aus dem Konstruktor aufgerufen. So wird eine Referenz auf den Thumb übergeben sowie die gewünschte Cursorform.

```
//Funktion BuildAdornerCorner. Aufrufendes Element wird auf cornerThumb
//übergeben ebenso wie gewünschter Cursor
void BuildAdornerCorner(ref Thumb cornerThumb, Cursor customizedCursor)
{
    //Bricht ab, wenn kein cornerThumb vorhanden
    if (cornerThumb != null) return;
    //Legt neues Element cornerThumb als Thumb an
    cornerThumb = new Thumb();

    //Legt das Aussehen des cornerThumb fest
    cornerThumb.Cursor = customizedCursor;
    cornerThumb.Height = cornerThumb.Width = 10;
    cornerThumb.Opacity = 0.70;
    cornerThumb.Background = new SolidColorBrush(Colors.Red);
    //Fügt das zuvor definierte Aussehen der visualCollection hinzu
    visualChildren.Add(cornerThumb);
}
```

Abb. 113: Erzeugen eines einzelnen Thumb

Nach Erzeugen eines neuen Thumb wird diesem das entsprechende Aussehen zugewiesen, so u.a. die definierte Cursorform, die Breite und Höhe sowie eine Transparenz und Füllfarbe.

Danach wird dieser Thumb der `VisualCollection` `visualChildren` zugewiesen. Eine `VisualCollection` ist eine Sammlung an `Visual`-Objekten. Als solche werden in der WPF alle Elemente bezeichnet, die auf dem Bildschirm sichtbar sind, bzw. sichtbar gemacht werden können. `Visuals` stellen wesentliche Merkmale eines Bildelements bereit, wie z.B. Darstellung, Kollisionsüberprüfung und Koordinatentransformation. Im wesentlichen also alle Aufgaben, die für das Generieren von Anfassern zum interaktiven Verändern von Objekten hier notwendig sind.

Für eine detaillierte Beschreibung des `Visual`-Prinzips und seiner Anwendung wird auf das Microsoft Developer Network (MSDN) unter [MS-02] verwiesen.

Über den Anfasser mittig auf der Stirnseite des idealisierten Profilstabs soll das für CAD-Systeme typische Verlängern/Verkürzen eines Stabes programmtechnisch nachgewiesen werden.

Im Konstruktor der Klasse `ResizingAdorner` wurde bei jedem Thumb das Ereignis `DragDelta` um einen selbstdefinierten Eventhandler ergänzt. Entsprechend der Position des Thumb wird dieser z.B. `HandleTop` genannt und dem Thumb `Top` zugewiesen.

Wird nun `Top` bewegt (`drag`), erkennt der Eventhandler `DragDeltaEventHandler` die Verschiebung und führt z.B. die Funktion `HandleTop` aus, siehe Bild 114.

```
void HandleTop(object sender, DragDeltaEventArgs args)
{
    //Casten von adornedElement
    FrameworkElement adornedElement = this.AdornedElement as FrameworkElement;
    //Casten von hitThumb
    Thumb hitThumb = sender as Thumb;
    //AdornerElement als rec
    Rectangle rec = AdornedElement as Rectangle;

    //Randabstand wird ermittelt
    Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top +
args.VerticalChange, 0, 0);
    adornedElement.Margin = dicke;
    rec.Height = Math.Max(rec.Height - args.VerticalChange,
hitThumb.DesiredSize.Height);
}
}
```

Abb. 114: Selbstdefinierter Eventhandler für Anfasser oben mittig zum Verlängern/Verkürzen des idealisierten Stabelements

Um das Element in der Größe verändern zu können, wird die Höhe des `adornedElement` verändert. Dazu wird zunächst das `adornedElement` als `rec` in der Funktion verfügbar gemacht. Der `sender`, also der Thumb `bottomRight`, wird als `hitThumb` in der Funktion verwendet.

Die Höhe des Elements setzt sich somit zusammen aus der aktuellen Höhe des `adornedElement` (`rec.Height`) zu der die Änderung der Mausposition in vertikaler Richtung (`args.VerticalChange`) addiert wird.

Die Funktion `Math.Max`, innerhalb der diese Argumente platziert sind, hat die Größe des Thumb (`hitThumb.desiredSize`) als zweites Argument. `Math.Max` berechnet aus diesen beiden Argumenten das jeweils größere und gibt diesen Wert zurück. Damit wird ausgeschlossen, dass das `adornedElement` kleiner skaliert werden kann, als das

Thumb groß ist. So wird verhindert, dass beim Skalieren negative Breiten und Höhen entstehen und das Programm abstürzt. Die Größe des Anfassers ist also die kleinste Größe, auf die ein Element verkleinert werden kann.

In einer dauerhaften Schleife, solange wie die Thumbs angezeigt werden, wird so jede Veränderung der Mausposition die Größe des Objektes verändern. Die flüssige Bewegung beim Skalieren ist also vielmehr ein ständiges Vergrößern (oder Verkleinern) des Objektes im Pixelbereich.

Typischerweise verändert das Ziehen an einem Anfasser die Ecke oder Kante eines Objektes, an der der Anfasser platziert ist. Zieht der Benutzer also am mittleren Anfasser an der oberen Stirnkante wird das Element nach oben vergrößert. Die untere Stirnkante bleibt dabei unverändert.

Bei `HandleTop`, dem mittleren Anfasser an der oberen Stirnkante, ist somit der Ursprung des Objekts zu beachten. So wird beim Skalieren die Höhe des Rechtecks verändert. Allerdings würde, da das Element über die linke obere Ecke als lokaler Nullpunkt auf dem Bildschirm ausgerichtet ist, ein Ziehen am mittleren, oberen Anfasser nicht die obere Kante verschieben, sondern das Element nach unten verlängern. Ein nicht gewünschter Effekt.

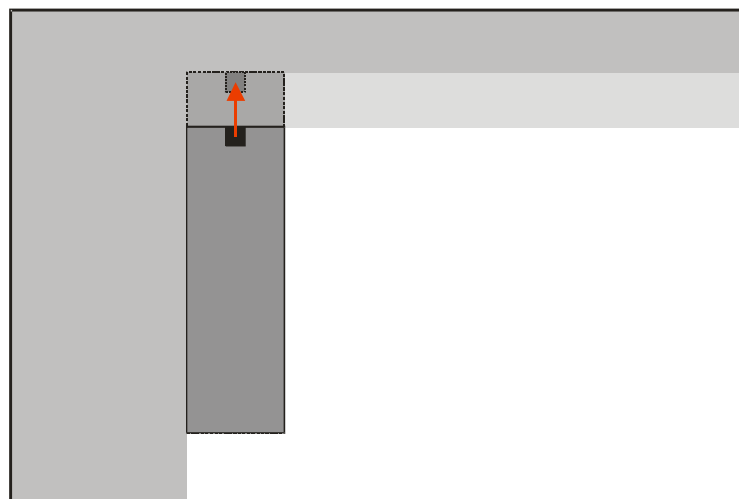


Abb. 115: Verschieben des mittleren oberen Referenzpunktes

Daher muss die Position des Elements in dem gleichen Maße verändert werden, wie das Element skaliert wird, siehe Bild 115. Dies geschieht indem der obere Randabstand (`Margin.Top`) verändert wird (siehe Bild 116).

```
//Verändern der Dicke des Randes um den Wert, den die Höhe verändert wird
Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top +
args.VerticalChange, 0, 0);
```

Abb. 116: Auszug aus `HandleTop`

Über die Zuweisung einer neuen Dicke, bei der der Rand links, oben, rechts und unten angegeben wird und danach dem `adornedElement` zugewiesen wird, wird das Element nach oben vergrößert. Programmtechnisch wird das Element hingegen nach unten vergrößert und gleichzeitig nach oben verschoben.

Auch dies geschieht in einer Schleife solange der Benutzer den Anfasser bewegt und wirkt wie eine einzige, flüssige Bewegung.

Für die anderen Anfasser in den Ecken und Mittelpunkten der Außenkanten wird entsprechend fortgefahren. Dabei sind die Argumente für das Verändern des Randes sowie der Berechnung der Höhe und Breite auf den entsprechenden Anfasser anzupassen.

Im Vergleich zu den in dieser Arbeit diskutierten CAD-Hochleistungssystemen BOCAD-3D und Tekla Structures ist dies ein äußerst effizientes, da direktes Verfahren. Mit den Anfassern an der oberen oder unteren Stirnkante ändert er die Länge, mit den Anfassern an den Seitenkanten kann er die Breite, also das Profil des Stabes, verändern.

Bei BOCAD-3D muss er dazu zunächst umständlich den Menübefehl „Detail – Verlängern/Verkürzen“ auswählen und in einem entsprechenden Dialog einen numerischen Wert eingeben. Nach Bestätigen des Dialogfeldes über „OK“ wählt der Benutzer den Stab aus. Nun muss er mit einem weiteren Mausklick die Stirnkante auswählen, an der die Änderung ausgeführt werden soll. In diese Richtung wird der Stab dann verlängert bzw. verkürzt. Ein äußerst kompliziertes für den Benutzer nicht effizientes Verfahren.

In Tekla ist dies nur unwesentlich besser gelöst. Konstruktionsobjekte besitzen einen Anfangs- und Endpunkt, die durch rosa Punkte auf dem Bildschirm dargestellt werden. Sie sind allerdings nicht direkt manipulierbar. Um die Länge des Stabes zu ändern, muss

der Benutzer den Start- oder Endpunkt mit der rechten Maustaste anklicken und im Kontextmenü „Verschieben“ auswählen. Danach kann er die neue Position für den Punkt festlegen wodurch die Länge des Stabes verändert wird. Auch dieses Verfahren ist ineffizient und für den Benutzer nicht direkt verständlich.

Das Aussehen der in diesem Beispiel entwickelten Anfasser muss gestaltbar sein, um selbsterklärende Anfasser zu ermöglichen, wie u.a. in [Chan02] vorgeschlagen und in [Azad07] fortentwickelt. In der bisher vorgestellten Basisversion wurden die Anfasser lediglich von der .NET-Klasse `Thumb` abgeleitet und boten daher nur wenig Möglichkeiten, ihr Aussehen zu verändern. Lediglich die Farbe und die Transparenz konnte variiert werden.

Über den programmtechnischen Ansatz, die `GetVisualChild`-Methode zu überschreiben, kann man die Anfasserelemente vollkommen frei gestalten, wie Bild 117 zeigt.

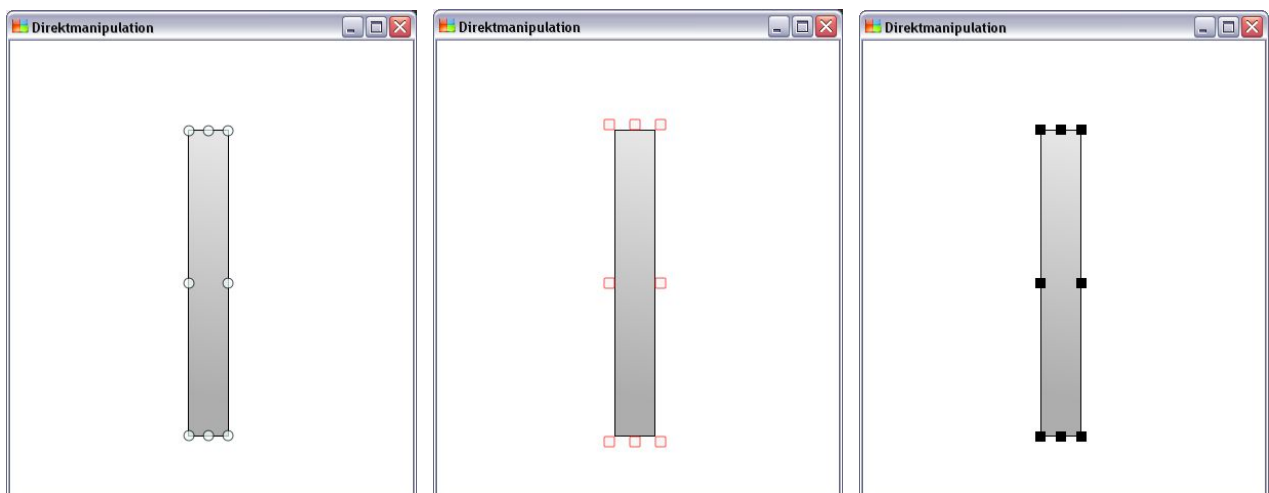


Abb. 117: Alternative Anfasserformen

Dazu wird eine neue Klasse `FlatThumb` erstellt, die von `Thumb` abgeleitet wird. Sie überschreibt die Methode `GetVisualChild` und gibt `null` (also nichts) zurück. Dadurch wird das `Thumb` quasi „unsichtbar“.

Die ebenfalls überschriebene Methode `OnRender` definiert nun das Erscheinungsbild. Es wird ein abgerundetes Rechteck gezeichnet, dessen Parameter variiert werden können: Füllfarbe, Randfarbe sowie die Eckabrundungen in X- und Y-Richtung. Ein kleiner

Wert rundet die Ecke nur leicht ab, ein hoher Wert (z.B. 5, das ist die halbe Höhe des Anfassers) macht aus dem abgerundeten Rechteck einen symmetrischen Kreis.

```
//Vererben der Klasse Thumb auf FlatThumb
class FlatThumb : Thumb
{
//Überschreiben der Methode GetVisualChild
protected override Visual GetVisualChild(int index)
{
return null;
}
//Überschreiben der Methode OnRender
protected override void OnRender(DrawingContext drawingContext)
{
//Zeichnen eines Rechtecks anstelle des normalen Thumbs
drawingContext.DrawRoundedRectangle(Brushes.WhiteSmoke, new Pen
(Brushes.Red, 1), new Rect(this.DesiredSize), 1, 1);
}
}
}
```

Abb. 118: Klasse FlatThumb für selbstdefinierte Anfasser

Das hier vorgestellte Prinzip der Adorner, also eines Schmuckelementes, lässt sich nicht nur für Anfasserelemente nutzen, die ein Element unmittelbar verändern. Auf einem solchen Adorner lassen sich sämtliche Elemente einer Benutzungsoberfläche platzieren. So bilden Adorner auch die programmtechnische Basis für die Forderungen in [Weck07]. Informationen zur Überwachung von vernetzt-kooperativen Konstruktionsprozessen, wie z.B. Symbole zur Darstellung der Freigabe, lassen sich so auf einem Adorner platzieren und anzeigen.

Die Notwendigkeit, dass die für intuitive Benutzungsoberflächen notwendige Programmierleistung nicht von Entwicklern in Softwarehäusern des Bauwesens allein und für jedes Produkt erneut erbracht werden kann, zeigt die Analyse der gängigen Programme der Marktführer und deren Schwächen in Kapitel 2.

Dieses Programmbeispiel demonstriert den programmtechnischen Ansatz, wie Thumbs die entscheidende Basis zur Schaffung einer direkten und dadurch effizienten Benutzungsoberfläche selbst für kleine Softwarehäuser bieten. Die in dieser Arbeit vorgestellte Klasse ResizingAdorner ist in dieser Form auf jegliche Objekte anwendbar, wobei Form und Aussehen der Anfasser der jeweiligen Programmumgebung angepasst werden können.

Fortentwickelt lässt sich dieser programmtechnische Ansatz nun um weitere Funktionen erweitern, so z.B. um das Kopieren eines Objektes mittels Ziehen bei gedrückter linker Maustaste und STRG-Taste oder um den in Kapitel 2 analysierten intelligenten Objektfang, bei dem das aktuelle Objekt beim Ausrichten am Nachbarn „magnetisch“ angezogen wird.

Dabei wird der Programmentwickler allerdings nicht auf ähnliche Weise wie bei den Thumbs unterstützt. Die Methoden müssen einzeln vom Programmierer entwickelt und implementiert werden.

Ein entsprechender Ansatz für den intelligenten Objektfang konnte bislang aufgrund der noch nicht kompletten, umfassenden Dokumentation aller neuen Funktionen des .NET-Frameworks 3.0 nicht entwickelt werden. Hierzu ist während des Verschiebens des Anfassers eine permanente Abfrage der unter der Maus befindlichen Objekte notwendig.

Der programmtechnische Ansatz für das Kopieren eines Objektes mittels Mausbewegung und gedrückter STRG-Taste konnte bereits erfolgreich nachgewiesen werden und befindet sich im Anhang dieser Arbeit. Der Quellcode kann so in jegliche weitere Softwareprodukte für das Bauwesen übernommen werden.

6 **Ausblick, weitere Forschung**

Die Kategorie der Bausoftware bietet in besonderem Maß die Chance, einen Standard für eine sprach- und textfreie Benutzungsoberfläche zu schaffen, die durch direkte Manipulation weltweit verständlich und durch Anfassersymbole am Objekt effizient ist. Schon [Chan02] stellte fest, dass die standardisierten Bauzeichnungen und Darstellungen ein großes Potential bieten, internationale Icons und Symbole zu entwickeln um auf Sprache in der Benutzungsoberfläche weitgehend zu verzichten.

Systemkonforme Menüs und Symbolleisten, die den Benutzer schnell und intuitiv Befehle finden und ausführen lassen sind bereits Standard in modernen grafischen Benutzungsoberflächen. Sie fehlen aber weitgehend in den fachspezifischen Programmen des Bauwesens, wie Analysen führender Bausoftware zeigten.

Anwendungen so zu konzipieren, dass sie selbst für kleine Softwarehäuser für einen internationalen Markt wirtschaftlich entwickelt und lokalisiert werden können, schafft die Basis für deren erfolgreiches Wachstum und Export.

Die direkte Manipulation durch Anfasser, direkt am Objekt platziert, macht Bausoftware international ohne Sprache intuitiv verständlich. Doch nur durch entsprechende Programmierwerkzeuge und Entwicklungsumgebungen wird dies mit dem hier entwickelten programmtechnischen Ansatz für kleine Softwareunternehmen des Bauwesens wirtschaftlich möglich.

Die vorgelegte Arbeit zeigt, wie zukünftige Benutzungsoberflächen speziell im Bauwesen für Kooperationen in weltweiten Netzen gestaltet werden müssen und bietet dazu den entsprechenden programmtechnischen Ansatz.

Anhand von gängiger Standardsoftware wird der aktuelle Ist-Zustand analysiert und entsprechende Empfehlungen gegeben. Eine Analyse der 3D-CAD-Hochleistungssysteme der Marktführer zeigt deren Stärken und Schwächen auf, die zukünftig gelöst werden müssen, speziell im Hinblick auf Systemkonformität und Benutzererwartung.

Dabei zeigt diese Arbeit, wie das Problem von nicht vermeidbaren Betextungen durch spezielle Werkzeuge zur Übersetzung gelöst wird und entwickelt aus diesem Ansatz eine effiziente und innovative Lösung zur Gestaltung von textfreien Menüs und Benutzungsoberflächen, wie sie in [Azad07] entwickelt werden.

Die in dieser Arbeit entwickelten Ansätze zum direkten Skalieren von Objekten zeigen, wie eine Benutzungsoberfläche sprachfrei ohne Texte intuitiv nutzbar ist – der programmtechnische Ansatz dieser Arbeit lässt sich dabei in weiteren Softwareprodukten nutzen. Er dient dabei nicht nur zur Effizienzsteigerung bei den hier vorgestellten CAD-Hochleistungssystemen BOCAD-3D und Tekla Structures, sondern kann für jegliche Software, z.B. AutoCAD, Inspiration und Anregung liefern.

Darüber hinaus gibt diese Arbeit einen Einstieg in die Welt der Grafikprogrammierung sowie in die Analyse von Benutzungsoberflächen. Denn trotz einer schier unbegrenzten Auswahl an Lehrbüchern zur Programmierung, Softwareanalyse und -entwicklung findet sich nur wenig Lektüre zum Erlernen der Grafikprogrammierung und der Programmierung von systemkonformen Benutzungsoberflächen.

Diese Arbeit zeigt mit der vorgestellten .NET-Klasse „Adorner“ die entsprechende Lösung, wie jegliche andere Elemente einer Benutzungsoberfläche direkt am zu bearbeitenden Objekt platziert werden können. So können auf einem solchen Adorner Symbole zur direkten Anzeige des Freigabestatus platziert werden, wie z.B. eine Verkehrsampel mit roten, gelben und grünen Symbolen.

Sie bildet so die programmtechnische Basislösung für die Zukunftsentwicklung einer Benutzungsoberfläche für das Bauwesen, wie sie in [Azad07] und [Weck07] konzipiert werden soll und in weiteren Master- und Diplomarbeiten entwickelt werden kann.

Literaturverzeichnis

- Appl-01 Apple Computer, Inc.
Apple Human Interface Guidelines
<http://developer.apple.com/documentation/UserExperience/>
Stand: Juni 2006
- Azad07 Hamid Azadnia
Konzeption einer zeichnungs-basierten, möglichst sprach- und textfreien
CAD-Benutzungsoberfläche im Bauwesen
Geplante Dissertation an der Bergischen Universität Wuppertal, 2007
- Balz00 Helmut Balzert
Lehrbuch der Software-Technik, Band 1
Spektrum Akademischer Verlag GmbH, 2000, ISBN 3-8274-0480-0
- Bann99 Gabriele Bannert & Martin Weitzel
Objektorientierter Softwareentwurf mit UML
Addison-Wesley, 1999, ISBN 3-8273-1487-9
- Blanc01 Jasmin Blanchette, Mark Summerfield
C++ GUI-Programmierung mit Qt3
Addison-Wesley, 2004, ISBN 3-8273-2348-7
- Booc94 Grady Booch
Objektorientierte Analyse und Design
Addison Wesley, 1994, ISBN 3-89319-673-0
- Chan02 Yongyu Chang
Eine von Landessprachen unabhängige Nutzoberfläche mit intelligenten
CAD-Objekten des Bauwesens
Dissertation an der Bergischen Universität Wuppertal, 2002
- EN-9241 CEN, Europäisches Institut für Normung
EN ISO 9241, Ergonomische Anforderungen für Bürotätigkeiten mit Bild-
schirmgeräten
DIN Deutsches Institut für Normung, 2001
- Gebb04 Norbert Gebbeken
Wörterbuch Bauwesen - konstruktiver Ingenieurbau
Berichte aus dem konstruktiven Ingenieurbau, Universität der Bundes-
wehr München, 2004
- GUI-01 Graphical User Interface Gallery
<http://www.aresluna.org/guidebook/>
Stand: Mai 2005

- Hell05 Günther Hellbardt
Software-Ergonomie
http://www1.informatik.uni-jena.de/Lehre/SoftErg/swe_001.htm
Stand: Mai 2005
- Hero01 Helmut Herold
Das Qt-Buch
Suse Press, 2001, ISBN 3-934678-76-9
- Hübn05 Rayan Abdullah, Roger Hübner
Piktogramm und Icons
Hermann Schmitz Mainz, 2005, ISBN 3-87439-649-5
- Kano95 Nadine Kano
Developing International Software
Microsoft Press, 1995, ISBN 1-55615-840-8
- Lehn99 Burkhard Lehner
KDE- und Qt-Programmierung
Addison-Wesley, 1999, ISBN 3-8273-1477-1
- MS-01 Microsoft Developer Network (MSDN)
User Interface Design and Development
<http://msdn.microsoft.com/ui/>
Stand: Juni 2006
- MS-02 Microsoft Developer Network (MSDN)
[http://msdn2.microsoft.com/en-us/library/
system.windows.media.visual.aspx](http://msdn2.microsoft.com/en-us/library/system.windows.media.visual.aspx)
Stand: Februar 2007
- Pege04 Georg Pegels
Grundlagen vernetzt-kooperativer Planungsprozesse für Komplettbau
Zwischenbericht DFG-Schwerpunktprogramm 1103, Bergische Universität Wuppertal, 2004
- Pege06 Georg Pegels, Torsten Weckmann
Allgemeingültige Benutzungsoberfläche für rechnergestützt koordinierte,
vernetzt-kooperative Planungsprozesse
DFG-Forschungsbericht, 2006
- Pege97 Georg Pegels, Paul Kutsch
Wettbewerbsfähigkeit und CAD-Konstruktion
Stahlbau, Heft 5, 1997, ISSN 0038-9145-25

- Petz00 Charles Petzold
Windows-Programmierung
Microsoft Press Deutschland, 2000, ISBN 3-86063-188-8
- Petz06 Charles Petzold
Anwendung = Code + Markup
Microsoft Press Deutschland, 2006, ISBN 3-86645-407-4
- Qt-Doc Trolltech AS
Qt Reference Documentation (Open Source Edition)
<http://doc.trolltech.com>
Stand: Februar 2007
- Qt-EaD Trolltech AS
Qt Examples and Demos (in Qt-Entwicklungsumgebung enthalten)
- Rask00 Jef Raskin
Das intelligente Interface
Addison-Wesley, 2000, ISBN 3-8273-1796-7
- Rieg05 Frank Rieg
Grafikprogrammierung für Windows
Hanser Verlag, 2005, ISBN 3-446-40009-5
- Rieß06 Thomas Rießinger
Informatik für Ingenieure und Naturwissenschaftler
Springer Verlag, 2006, ISBN 3-540-26243-1
- Weck07 Torsten Weckmann
Allgemeingültige Benutzungsoberfläche für überwachte, vernetzt-kooperative Planungsprozesse
Geplante Dissertation an der Bergischen Universität Wuppertal, 2007
- WIKI-API http://de.wikipedia.org/wiki/Application_Programming_Interface
Stand: Dezember 2006
- WIKI-BASIC <http://de.wikipedia.org/wiki/BASIC>
Stand: Dezember 2006
- WIKI-C http://de.wikipedia.org/wiki/C_Programmiersprache
Stand: Dezember 2006
- WIKI-C# <http://de.wikipedia.org/wiki/C-Sharp>
Stand: Dezember 2006
- WIKI-C++ <http://de.wikipedia.org/wiki/C-Plusplus>
Stand: Dezember 2006

WIKI-Fortran	http://de.wikipedia.org/wiki/FORTRAN Stand: Dezember 2006
WIKI-GUI	http://de.wikipedia.org/wiki/Grafische_Benutzeroberflaeche Stand: Dezember 2006
WIKI-Hallo	http://de.wikipedia.org/wiki/Hallo-Welt-Programm Stand: Dezember 2006
WIKI-Java	http://de.wikipedia.org/wiki/Java_Programmiersprache Stand: Dezember 2006
WIKI-JS	http://de.wikipedia.org/wiki/Javascript Stand: Dezember 2006
WIKI-Markup	http://de.wikipedia.org/wiki/Markup Stand: Dezember 2006
WIKI-Maus	http://de.wikipedia.org/wiki/Computermaus Stand: März 2005
WIKI-MS	http://de.wikipedia.org/wiki/Microsoft Stand: Dezember 2006
WIKI-Qt	http://de.wikipedia.org/wiki/Qt_Toolkit Stand: Dezember 2006
WIKI-VB	http://de.wikipedia.org/wiki/Visual_Basic Stand: Dezember 2006
WIKI-VB.NET	http://de.wikipedia.org/wiki/VB.NET Stand: Dezember 2006
WIKI-Win3.x	http://de.wikipedia.org/wiki/Microsoft_Windows_3.x Stand: Dezember 2006
WIKI-XML	http://de.wikipedia.org/wiki/XML Stand: Dezember 2006
WWW-C#	Golo Haas Guide to C# http://www.guidetocsharp.de Stand: April 2005

Anhang

Software-Prototyp I – Menü- und Symbolleisten am Beispiel eines Texteditors

main.cpp

```
#include <QApplication>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    Q_INIT_RESOURCE(application);

    QApplication app(argc, argv);
    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

mainwindow.cpp

```
#include <QtGui>
#include <QFont>

#include "mainwindow.h"

MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);
    QFont f("Courier", 12, QFont::Bold);
    setFont(f);

    createAction();
    createMenus();
    createToolBars();
    createStatusBar();

    readSettings();

    connect(textEdit->document(), SIGNAL(contentsChanged()),
           this, SLOT(documentWasModified()));

    setCurrentFile("");
}

void MainWindow::closeEvent(QCloseEvent *event)
{
    if (maybeSave()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

```

void MainWindow::newFile()
{
    if (maybeSave()) {
        textEdit->clear();
        setCurrentFile("");
    }
}

void MainWindow::open()
{
    if (maybeSave()) {
        QString fileName = QFileDialog::getOpenFileName(this);
        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}

bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this);
    if (fileName.isEmpty())
        return false;

    return saveFile(fileName);
}

void MainWindow::about()
{
    QMessageBox::about(this, tr("About Application"),
        tr("The <b>Application</b> example demonstrates how to "
            "write modern GUI applications using Qt, with a menu bar, "
            "toolbars, and a status bar."));
}

void MainWindow::documentWasModified()
{
    setWindowModified(textEdit->document()->isModified());
}

void MainWindow::createActions()
{
    newAct = new QAction(QIcon(":/images/new.png"), tr("&New"), this);
    newAct->setShortcut(tr("Ctrl+N"));
    newAct->setStatusTip(tr("Create a new file"));
    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

    openAct = new QAction(QIcon(":/images/open.png"), tr("&Open..."),
        this);
    openAct->setShortcut(tr("Ctrl+O"));
}

```

```

openAct->setStatusTip(tr("Open an existing file"));
connect(openAct, SIGNAL(triggered()), this, SLOT(open()));

saveAct = new QAction(QIcon(":/images/save.png"), tr("&Save"), this);
saveAct->setShortcut(tr("Ctrl+S"));
saveAct->setStatusTip(tr("Save the document to disk"));
connect(saveAct, SIGNAL(triggered()), this, SLOT(save()));

saveAsAct = new QAction(tr("Save &As..."), this);
saveAsAct->setStatusTip(tr("Save the document under a new name"));
connect(saveAsAct, SIGNAL(triggered()), this, SLOT(saveAs()));

exitAct = new QAction(tr("E&xit"), this);
exitAct->setShortcut(tr("Ctrl+Q"));
exitAct->setStatusTip(tr("Exit the application"));
connect(exitAct, SIGNAL(triggered()), this, SLOT(close()));

cutAct = new QAction(QIcon(":/images/cut.png"), tr("Cu&t"), this);
cutAct->setShortcut(tr("Ctrl+X"));
cutAct->setStatusTip(tr("Cut the current selection's contents to the "
    "clipboard"));
connect(cutAct, SIGNAL(triggered()), textEdit, SLOT(cut()));

copyAct = new QAction(QIcon(":/images/copy.png"), tr("&Copy"), this);
copyAct->setShortcut(tr("Ctrl+C"));
copyAct->setStatusTip(tr("Copy the current selection's contents to the "
    "clipboard"));
connect(copyAct, SIGNAL(triggered()), textEdit, SLOT(copy()));

pasteAct = new QAction(QIcon(":/images/paste.png"), tr("&Paste"),
this);
pasteAct->setShortcut(tr("Ctrl+V"));
pasteAct->setStatusTip(tr("Paste the clipboard's contents into the
current "
    "selection"));
connect(pasteAct, SIGNAL(triggered()), textEdit, SLOT(paste()));

aboutAct = new QAction(tr("&About"), this);
aboutAct->setStatusTip(tr("Show the application's About box"));
connect(aboutAct, SIGNAL(triggered()), this, SLOT(about()));

aboutQtAct = new QAction(tr("About &Qt"), this);
aboutQtAct->setStatusTip(tr("Show the Qt library's About box"));
connect(aboutQtAct, SIGNAL(triggered()), qApp, SLOT(aboutQt()));

cutAct->setEnabled(false);
copyAct->setEnabled(false);
connect(textEdit, SIGNAL(copyAvailable(bool)),
    cutAct, SLOT(setEnabled(bool)));
connect(textEdit, SIGNAL(copyAvailable(bool)),
    copyAct, SLOT(setEnabled(bool)));
}

void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAct);
    fileMenu->addAction(openAct);
}

```

```

fileMenu->addAction(saveAct);
fileMenu->addAction(saveAsAct);
fileMenu->addSeparator();
fileMenu->addAction(exitAct);

editMenu = menuBar()->addMenu(tr("&Edit"));
editMenu->addAction(cutAct);
editMenu->addAction(copyAct);
editMenu->addAction(pasteAct);

menuBar()->addSeparator();

helpMenu = menuBar()->addMenu(tr("&Help"));
helpMenu->addAction(aboutAct);
helpMenu->addAction(aboutQtAct);
}

void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(newAct);
    fileToolBar->addAction(openAct);
    fileToolBar->addAction(saveAct);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(cutAct);
    editToolBar->addAction(copyAct);
    editToolBar->addAction(pasteAct);
}

void MainWindow::createStatusBar()
{
    statusBar()->showMessage(tr("Ready"));
}

void MainWindow::readSettings()
{
    QSettings settings("Trolltech", "Application Example");
    QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
    QSize size = settings.value("size", QSize(400, 400)).toSize();
    resize(size);
    move(pos);
}

void MainWindow::writeSettings()
{
    QSettings settings("Trolltech", "Application Example");
    settings.setValue("pos", pos());
    settings.setValue("size", size());
}

bool MainWindow::maybeSave()
{
    if (textEdit->document()->isModified()) {
        int ret = QMessageBox::warning(this, tr("Application"),
            tr("The document has been modified.\n"
                "Do you want to save your changes?"),
            QMessageBox::Yes | QMessageBox::Default,
            QMessageBox::No,

```

```

        QMessageBox::Cancel | QMessageBox::Escape);
    if (ret == QMessageBox::Yes)
        return save();
    else if (ret == QMessageBox::Cancel)
        return false;
    }
    return true;
}

void MainWindow::loadFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        QMessageBox::warning(this, tr("Application"),
            tr("Cannot read file %1:\n%2.")
                .arg(fileName)
                .arg(file.errorString()));
        return;
    }

    QTextStream in(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    textEdit->setPlainText(in.readAll());
    QApplication::restoreOverrideCursor();

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
}

bool MainWindow::saveFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::WriteOnly | QFile::Text)) {
        QMessageBox::warning(this, tr("Application"),
            tr("Cannot write file %1:\n%2.")
                .arg(fileName)
                .arg(file.errorString()));
        return false;
    }

    QTextStream out(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    out << textEdit->toPlainText();
    QApplication::restoreOverrideCursor();

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}

void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    textEdit->document()->setModified(false);
    setWindowModified(false);

    QString shownName;
    if (curFile.isEmpty())
        shownName = "untitled.txt";
}

```

```

    else
        shownName = strippedName(curFile);

    setWindowTitle(tr("%1[*] - %2").arg(shownName).arg(tr("List-Editor")));
}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}

```

mainwindow.h

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QMenu;
class QTextEdit;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void about();
    void documentWasModified();

private:
    void createActions();
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool maybeSave();
    void loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);

    QTextEdit *textEdit;
    QString curFile;

    QMenu *fileMenu;
    QMenu *editMenu;
    QMenu *helpMenu;
    QToolBar *fileToolBar;
}

```

```
    QToolBar *editToolBar;
    QAction *newAct;
    QAction *openAct;
    QAction *saveAct;
    QAction *saveAsAct;
    QAction *exitAct;
    QAction *cutAct;
    QAction *copyAct;
    QAction *pasteAct;
    QAction *aboutAct;
    QAction *aboutQtAct;
};

#endif
```

Software-Prototyp II – Automatische Übersetzung von unvermeidbaren Texten

main.cpp

```
#include <QApplication>
#include <QTranslator>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    Q_INIT_RESOURCE(application);

    QApplication app(argc, argv);

    QTranslator translator;
    translator.load("application_de");
    app.installTranslator(&translator);

    Qt::LayoutDirection layoutDirection (Qt::RightToLeft);

    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

mainwindow.cpp und mainwindow.h entsprechen denen aus Software-Prototyp I. Die Übersetzungsdatei wird hier aufgrund des XML-Formats nicht dargestellt, diese befindet sich auf der beiliegenden CD und sollte mit QtLinguist geöffnet werden.

Software-Prototyp II Erweiterung – Dynamischer Sprachwechsel

main.cpp

```
#include <QApplication>
#include <QTranslator>

#include "mainwindow.h"
```

```
int main(int argc, char *argv[])
{
    Q_INIT_RESOURCE(application);

    QApplication app(argc, argv);

    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

mainwindow.cpp

```
#include <QtGui>
#include <QFont>
#include <QApplication>
#include <QTranslator>
#include <QActionGroup>
#include <QMenu>

#include "mainwindow.h"

MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);
    //DS
    QFont f("Courier", 12, QFont::Bold);
    setFont(f);

    qApp->installTranslator(&appTranslator);
    qApp->installTranslator(&qtTranslator);
    qmPath=qApp->applicationDirPath()+"/translations";

    createAction();
    createMenus();
    createToolBars();
    createStatusBar();
    retranslateUi();

    readSettings();

    connect(textEdit->document(), SIGNAL(contentsChanged()),
           this, SLOT(documentWasModified()));

    setCurrentFile("");
}

void MainWindow::closeEvent(QCloseEvent *event)
{
    if (maybeSave()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

```

}

void MainWindow::newFile()
{
    if (maybeSave()) {
        textEdit->clear();
        setCurrentFile("");
    }
}

void MainWindow::open()
{
    if (maybeSave()) {
        QString fileName = QFileDialog::getOpenFileName(this);
        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}

bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this);
    if (fileName.isEmpty())
        return false;

    return saveFile(fileName);
}

void MainWindow::about()
{
    QMessageBox::about(this, tr("About Application"),
        tr("The <b>Application</b> example demonstrates how to "
            "write modern GUI applications using Qt, with a menu bar,
            "
            "toolbars, and a status bar."));
}

void MainWindow::documentWasModified()
{
    setWindowModified(textEdit->document()->isModified());
}

void MainWindow::createActions()
{
    newAct = new QAction(this);
    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

    openAct = new QAction(this);

    connect(openAct, SIGNAL(triggered()), this, SLOT(open()));
}

```

```

saveAct = new QAction(this);
connect(saveAct, SIGNAL(triggered()), this, SLOT(save()));

saveAsAct = new QAction(this);
connect(saveAsAct, SIGNAL(triggered()), this, SLOT(saveAs()));

exitAct = new QAction(this);
connect(exitAct, SIGNAL(triggered()), this, SLOT(close()));

cutAct = new QAction(this);
connect(cutAct, SIGNAL(triggered()), textEdit, SLOT(cut()));

copyAct = new QAction(this);
connect(copyAct, SIGNAL(triggered()), textEdit, SLOT(copy()));

pasteAct = new QAction(this);
connect(pasteAct, SIGNAL(triggered()), textEdit, SLOT(paste()));

aboutAct = new QAction(this);
connect(aboutAct, SIGNAL(triggered()), this, SLOT(about()));

aboutQtAct = new QAction(this);
connect(aboutQtAct, SIGNAL(triggered()), qApp, SLOT(aboutQt()));

cutAct->setEnabled(false);
copyAct->setEnabled(false);
connect(textEdit, SIGNAL(copyAvailable(bool)),
        cutAct, SLOT(setEnabled(bool)));
connect(textEdit, SIGNAL(copyAvailable(bool)),
        copyAct, SLOT(setEnabled(bool)));
}

void MainWindow::retranslateUi()
{
    newAct->setIcon(QIcon(":/images/filenew.png"));
    newAct->setText(tr("&New"));
    newAct->setShortcut(tr("Ctrl+N"));
    newAct->setStatusTip(tr("Create a new file"));

    openAct->setIcon(QIcon(":/images/fileopen.png"));
    openAct->setText(tr("&Open..."));
    openAct->setShortcut(tr("Ctrl+O"));
    openAct->setStatusTip(tr("Open an existing file"));

    saveAct->setIcon(QIcon(":/images/filesave.png"));
    saveAct->setText(tr("&Save"));
    saveAct->setShortcut(tr("Ctrl+S"));
    saveAct->setStatusTip(tr("Save the document to disk"));

    saveAsAct->setIcon(QIcon(":/images/filesaveas.png"));
    saveAsAct->setText(tr("Save &As..."));
    saveAsAct->setStatusTip(tr("Save the document under a new name"));

    exitAct->setIcon(QIcon(":/images/filenew.png"));
    exitAct->setText(tr("E&xit"));
    exitAct->setShortcut(tr("Ctrl+Q"));
    exitAct->setStatusTip(tr("Exit the application"));
}

```

```

cutAct->setIcon(QIcon(":/images/editcut.png"));
cutAct->setText(tr("Cu&t"));
cutAct->setShortcut(tr("Ctrl+X"));
cutAct->setStatusTip(tr("Cut the current selections contents
to the clipboard"));

copyAct->setIcon(QIcon(":/images/editcopy.png"));
copyAct->setText(tr("&Copy"));
copyAct->setShortcut(tr("Ctrl+C"));
copyAct->setStatusTip(tr("Copy the current selection's contents to
the clipboard"));

pasteAct->setIcon(QIcon(":/images/editpaste.png"));
pasteAct->setText(tr("&Paste"));
pasteAct->setShortcut(tr("Ctrl+V"));
pasteAct->setStatusTip(tr("Paste the clipboard's contents into the
current selection"));

aboutAct->setIcon(QIcon(":/images/messagebox_info.png"));
aboutAct->setText(tr("&About"));
aboutAct->setStatusTip(tr("Show the application's About box"));

aboutQtAct->setIcon(QIcon(":/images/linguist.png"));
aboutQtAct->setText(tr("About &Qt"));
aboutQtAct->setStatusTip(tr("Show the Qt library's About box"));

fileMenu->setTitle(tr("&File"));
editMenu->setTitle(tr("&Edit"));
languageMenu->setTitle(tr("Language"));
helpMenu->setTitle(tr("&Help"));
}

void MainWindow::switchToLanguage(QAction *action)
{
    QString locale = action->data().toString();
    appTranslator.load("application_" + locale, qmPath);
    qtTranslator.load("qt_" + locale, qmPath);
    retranslateUi();
}

void MainWindow::createLanguageMenu()
{
    languageMenu=menuBar()->addMenu("&Language");

    languageActionGroup=new QActionGroup(this);
    connect(languageActionGroup, SIGNAL(triggered(QAction *)), this,
SLOT(switchToLanguage(QAction *)));

    QDir dir(qmPath);
    QStringList fileNames = dir.entryList(QStringList
("application_*.qm"));

    for (int i=0; i < fileNames.size(); ++i) {
        QString locale = fileNames[i];
        locale.remove(0,locale.indexOf('_')+1);
        locale.truncate(locale.lastIndexOf('.'));

        QTranslator translator;

```

```

        translator.load(fileNames[i], qmPath);
        QString language = translator.translate
("MainWindow", "English");

        QAction *action = new QAction(tr("%1 %2").arg(i+1).arg
(language), this);

        action->setCheckable(true);
        action->setData(locale);
        action->setIcon(QIcon("images/" + locale + ".png"));

        languageMenu->addAction(action);
        languageActionGroup->addAction(action);

        if(language=="English")
            action->setChecked(true);
    }
}

void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAct);
    fileMenu->addAction(openAct);
    fileMenu->addAction(saveAct);
    fileMenu->addAction(saveAsAct);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAct);

    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAct);
    editMenu->addAction(copyAct);
    editMenu->addAction(pasteAct);

    createLanguageMenu();

    menuBar()->addSeparator();

    helpMenu = menuBar()->addMenu(tr("&Help"));
    helpMenu->addAction(aboutAct);
    helpMenu->addAction(aboutQtAct);
}

void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(newAct);
    fileToolBar->addAction(openAct);
    fileToolBar->addAction(saveAct);

    editToolBar = addToolBar(tr("Edit"));
    editToolBar->addAction(cutAct);
    editToolBar->addAction(copyAct);
    editToolBar->addAction(pasteAct);
}

void MainWindow::createStatusBar()
{

```

```

        statusBar()->showMessage(tr("Ready"));
    }

void MainWindow::readSettings()
{
    QSettings settings("Trolltech", "Application Example");
    QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
    QSize size = settings.value("size", QSize(400, 400)).toSize();
    resize(size);
    move(pos);
}

void MainWindow::writeSettings()
{
    QSettings settings("Trolltech", "Application Example");
    settings.setValue("pos", pos());
    settings.setValue("size", size());
}

bool MainWindow::maybeSave()
{
    if (textEdit->document()->isModified()) {
        int ret = QMessageBox::warning(this, tr("Application"),
            tr("The document has been modified.\n"
              "Do you want to save your changes?"),
            QMessageBox::Yes | QMessageBox::Default,
            QMessageBox::No,
            QMessageBox::Cancel | QMessageBox::Escape);
        if (ret == QMessageBox::Yes)
            return save();
        else if (ret == QMessageBox::Cancel)
            return false;
    }
    return true;
}

void MainWindow::loadFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        QMessageBox::warning(this, tr("Application"),
            tr("Cannot read file %1:\n%2.")
                .arg(fileName)
                .arg(file.errorString()));
        return;
    }

    QTextStream in(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    textEdit->setPlainText(in.readAll());
    QApplication::restoreOverrideCursor();

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
}

bool MainWindow::saveFile(const QString &fileName)
{
    QFile file(fileName);

```

```

    if (!file.open(QFile::WriteOnly | QFile::Text)) {
        QMessageBox::warning(this, tr("Application"),
            tr("Cannot write file %1:\n%2.")
                .arg(fileName)
                .arg(file.errorString()));
        return false;
    }

    QTextStream out(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    out << textEdit->toPlainText();
    QApplication::restoreOverrideCursor();

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}

void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    textEdit->document()->setModified(false);
    setWindowModified(false);

    QString shownName;
    if (curFile.isEmpty())
        shownName = "untitled.txt";
    else
        shownName = strippedName(curFile);

    setWindowTitle(tr("%1[*] - %2").arg(shownName).arg(tr(
("Texteditor"))));
}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFile::FileInfo(fullFileName).fileName();
}

```

Software-Prototyp III – Elementares CAD-System mit sprach- und textfreier Benutzungsoberfläche

main.cpp

```

#include <qapplication.h>

#include "cadview.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Q3Canvas canvas;
    canvas.resize(640, 480);
    CadView view(&canvas);
    view.setCaption(QObject::tr("CAD-System"));
    app.setMainWidget(&view);
}

```

```
    view.show();
    return app.exec();
}
```

cadview.cpp

```
#include <qaction.h>
#include <qapplication.h>
#include <qclipboard.h>
#include <qinputdialog.h>
#include <qpainter.h>
#include <q3popupmenu.h>
#include <Q3PointArray>
#include <QContextMenuEvent>
#include <QMouseEvent>
#include <QTextIStream>
#include <Q3CanvasItem>

#include <algorithm>
using namespace std;

#include "cadview.h"
#include "propertiesdialog.h"

CadView::CadView(Q3Canvas *canvas, QWidget *parent,
                const char *name)
    : Q3CanvasView(canvas, parent, name)
{
    pendingItem = 0;
    activeItem = 0;
    minZ = 0;
    maxZ = 0;
    createActions();
}

void CadView::createActions()
{
    cutAct = new QAction(QIcon("images/cut.png"), tr("&Cut"), this);
    cutAct->setShortcut(tr("Ctrl+X"));
    cutAct->setStatusTip(tr("Cutting out marked range"));
    connect(cutAct, SIGNAL(activated()), this, SLOT(cut()));

    copyAct = new QAction(QIcon("images/copy.png"), tr("&Copy"), this);
    copyAct->setShortcut(tr("Ctrl+C"));
    copyAct->setStatusTip(tr("Copies marked range"));
    connect(copyAct, SIGNAL(activated()), this, SLOT(copy()));

    pasteAct = new QAction(QIcon("images/paste.png"), tr("&Paste"),
this);
    pasteAct->setShortcut(tr("Ctrl+V"));
    pasteAct->setStatusTip(tr("Pastes in"));
    connect(pasteAct, SIGNAL(activated()), this, SLOT(paste()));

    deleteAct = new QAction(QIcon("images/delete.png"), tr("&Delete"),
this);
    deleteAct->setShortcut(tr("Del"));
    deleteAct->setStatusTip(tr("Deletes"));
    connect(deleteAct, SIGNAL(activated()), this, SLOT(del()));
}
```



```

        propertiesAct = new QAction(tr("&Properties..."), this);
propertiesAct->setShortcut(tr("Del"));
propertiesAct->setStatusTip(tr("Properties"));
connect(propertiesAct, SIGNAL(activated()), this, SLOT(properties()));

        addBoxAct = new QAction(QIcon("images/box.png"), tr("Add &Box"),
this);
addBoxAct->setShortcut(tr("Del"));
addBoxAct->setStatusTip(tr("Adds box"));
connect(addBoxAct, SIGNAL(activated()), this, SLOT(addBox()));

        addLineAct = new QAction(QIcon("images/line.png"), tr("Add &Line"),
this);
addLineAct->setShortcut(tr("Del"));
addLineAct->setStatusTip(tr("Adds line"));
connect(addLineAct, SIGNAL(activated()), this, SLOT(addLine()));

        bringToFrontAct = new QAction(QIcon("images/bringtofront.png"), tr
("Bring to &Front"), this);
bringToFrontAct->setShortcut(tr("Del"));
bringToFrontAct->setStatusTip(tr("Adds box"));
connect(bringToFrontAct, SIGNAL(activated()), this, SLOT(bringToFront
()));

        sendToBackAct = new QAction(QIcon("images/sendtoback.png"), tr("Send
to &Back"), this);
bringToFrontAct->setShortcut(tr("Del"));
sendToBackAct->setStatusTip(tr("Adds box"));
connect(sendToBackAct, SIGNAL(activated()), this, SLOT(sendToBack()));
}

void CadView::contentsContextMenuEvent(QContextMenuEvent *event)
{
    Q3PopupMenu contextMenu(this);
    if (activeItem) {
        cutAct->addTo(&contextMenu);
        copyAct->addTo(&contextMenu);
        deleteAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        bringToFrontAct->addTo(&contextMenu);
        sendToBackAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        propertiesAct->addTo(&contextMenu);
    } else {
        pasteAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        addBoxAct->addTo(&contextMenu);
        addLineAct->addTo(&contextMenu);
    }
    contextMenu.exec(event->globalPos());
}

void CadView::addBox()
{
    addItem(new DiagramBox(canvas()));
}

void CadView::addLine()

```

```

{
    addItem(new DiagramLine(canvas()));
}

void CadView::addItem(Q3CanvasItem *item)
{
    delete pendingItem;
    pendingItem = item;
    setActiveItem(0);
    setCursor(Qt::crossCursor);
}

void CadView::contentsMouseEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton && pendingItem) {
        pendingItem->move(event->pos().x(), event->pos().y());
        showNewItem(pendingItem);
        pendingItem = 0;
        unsetCursor();
    } else {
        Q3CanvasItemList items = canvas()->collisions(event->pos());
        if (items.empty())
            setActiveItem(0);
        else
            setActiveItem(*items.begin());
    }
    lastPos = event->pos();
}

void CadView::contentsMouseMoveEvent(QMouseEvent *event)
{
    if (event->state() & Qt::LeftButton) {
        if (activeItem) {

            activeItem->moveBy(event->pos().x() - lastPos.x(),
                               event->pos().y() - lastPos.y());
            lastPos = event->pos();
            canvas()->update();
        }
    }
}

void CadView::contentsMouseDoubleClickEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton && activeItem
        && activeItem->rtti() == DiagramBox::RTTI) {
        DiagramBox *box = (DiagramBox *)activeItem;
        bool ok;

        QString newText = QInputDialog::getText(
            tr("Diagram"), tr("Enter new text:"),
            QLineEdit::Normal, box->text(), &ok, this);
        if (ok) {
            box->setText(newText);
            canvas()->update();
        }
    }
}

```

```

void CadView::bringToFront()
{
    if (activeItem) {
        ++maxZ;
        activeItem->setZ(maxZ);
        canvas()->update();
    }
}

void CadView::sendToBack()
{
    if (activeItem) {
        --minZ;
        activeItem->setZ(minZ);
        canvas()->update();
    }
}

void CadView::cut()
{
    copy();
    del();
}

void CadView::copy()
{
    if (activeItem) {
        QString str;

        if (activeItem->rtti() == DiagramBox::RTTI) {
            DiagramBox *box = (DiagramBox *)activeItem;
            str = QString("DiagramBox %1 %2 %3 %4 %5")
                .arg(box->width())
                .arg(box->height())
                .arg(box->pen().color().name())
                .arg(box->brush().color().name())
                .arg(box->text());
        } else if (activeItem->rtti() == DiagramLine::RTTI) {
            DiagramLine *line = (DiagramLine *)activeItem;
            QPoint delta = line->endPoint() - line->startPoint();
            str = QString("DiagramLine %1 %2 %3")
                .arg(delta.x())
                .arg(delta.y())
                .arg(line->pen().color().name());
        }
        QApplication::clipboard()->setText(str);
    }
}

void CadView::paste()
{
    QString str = QApplication::clipboard()->text();
    QTextIStream in(&str);
    QString tag;

    in >> tag;
    if (tag == "DiagramBox") {
        int width;
        int height;

```

```

    QString lineColor;
    QString fillColor;
    QString text;

    in >> width >> height >> lineColor >> fillColor;
    text = in.read();

    DiagramBox *box = new DiagramBox(canvas());
    box->move(20, 20);
    box->setSize(width, height);
    box->setText(text);
    box->setPen(QColor(lineColor));
    box->setBrush(QColor(fillColor));
    showNewItem(box);
} else if (tag == "DiagramLine") {
    int deltaX;
    int deltaY;
    QString lineColor;

    in >> deltaX >> deltaY >> lineColor;

    DiagramLine *line = new DiagramLine(canvas());
    line->move(20, 20);
    line->setPoints(0, 0, deltaX, deltaY);
    line->setPen(QColor(lineColor));
    showNewItem(line);
}
}

void CadView::del()
{
    if (activeItem) {
        Q3CanvasItem *item = activeItem;
        setActiveItem(0);
        delete item;
        canvas()->update();
    }
}

void CadView::properties()
{
    if (activeItem) {
        PropertiesDialog dialog;
        dialog.exec(activeItem);
    }
}

void CadView::showNewItem(Q3CanvasItem *item)
{
    setActiveItem(item);
    bringToFront();
    item->show();
    canvas()->update();
}

void CadView::setActiveItem(Q3CanvasItem *item)
{
    if (item != activeItem) {
        if (activeItem)

```

```

        activeItem->setActive(false);
        activeItem = item;
        if (activeItem)
            activeItem->setActive(true);
        canvas()->update();
    }
}

const int Margin = 2;

void drawActiveHandle(QPainter &painter, const QPoint &center)
{
    painter.setPen(Qt::black);
    painter.setBrush(Qt::gray);
    painter.drawRect(center.x() - Margin, center.y() - Margin,
                    2 * Margin + 1, 2 * Margin + 1);
}

DiagramBox::DiagramBox(Q3Canvas *canvas)
    : Q3CanvasRectangle(canvas)
{
    setSize(100, 60);
    setPen(QPen(Qt::black));
    setBrush(Qt::white);
    str = "Text";
}

DiagramBox::DiagramBox(Q3Canvas *canvas, long x, long y)
    : Q3CanvasRectangle(canvas)
{
    setSize(x, y);
    setPen(QPen(Qt::black));
    setBrush(Qt::white);
    str = "Text";
}

DiagramBox::~DiagramBox()
{
    hide();
}

void DiagramBox::setText(const QString &newText)
{
    str = newText;
    update();
}

void DiagramBox::drawShape(QPainter &painter)
{
    Q3CanvasRectangle::drawShape(painter);

    painter.drawText(rect(), Qt::AlignCenter, text());
    if (isActive()) {
        drawActiveHandle(painter, rect().topLeft());
        drawActiveHandle(painter, rect().topRight());
        drawActiveHandle(painter, rect().bottomLeft());
        drawActiveHandle(painter, rect().bottomRight());
    }
}

```

```

QRect DiagramBox::boundingRect() const
{
    return QRect((int)x() - Margin, (int)y() - Margin,
        width() + 2 * Margin, height() + 2 * Margin);
}

DiagramLine::DiagramLine(Q3Canvas *canvas)
    : Q3CanvasLine(canvas)
{
    setPoints(0, 0, 0, 99);
}

DiagramLine::~~DiagramLine()
{
    hide();
}

void DiagramLine::drawShape(QPainter &painter)
{
    Q3CanvasLine::drawShape(painter);
    if (isActive()) {
        drawActiveHandle(painter, startPoint() + offset());
        drawActiveHandle(painter, endPoint() + offset());
    }
}

Q3PointArray DiagramLine::areaPoints() const
{
    const int Extra = Margin + 1;
    Q3PointArray points(6);
    QPoint pointA = startPoint() + offset();
    QPoint pointB = endPoint() + offset();

    if (pointA.x() > pointB.x())
        swap(pointA, pointB);

    points[0] = pointA + QPoint(-Extra, -Extra);
    points[1] = pointA + QPoint(-Extra, +Extra);
    points[3] = pointB + QPoint(+Extra, +Extra);
    points[4] = pointB + QPoint(+Extra, -Extra);
    if (pointA.y() > pointB.y()) {
        points[2] = pointA + QPoint(+Extra, +Extra);
        points[5] = pointB + QPoint(-Extra, -Extra);
    } else {
        points[2] = pointB + QPoint(-Extra, +Extra);
        points[5] = pointA + QPoint(+Extra, -Extra);
    }
    return points;
}

```

cadview.h

```

#ifndef CADVIEW_H
#define CADVIEW_H

#include <q3canvas.h>
//Added by qt3to4:

```

```

#include <QContextMenuEvent>
#include <QMouseEvent>
#include <Q3PointArray>

class QAction;

class CadView : public Q3CanvasView
{
    Q_OBJECT
public:
    CadView(Q3Canvas *canvas, QWidget *parent = 0,
           const char *name = 0);

public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void properties();
    void addBox();
    void addLine();
    void bringToFront();
    void sendToBack();

protected:
    void contentsContextMenuEvent(QContextMenuEvent *event);
    void contentsMouseEvent(QMouseEvent *event);
    void contentsMouseMoveEvent(QMouseEvent *event);
    void contentsMouseDoubleClickEvent(QMouseEvent *event);

private:
    void createActions();
    void addItem(Q3CanvasItem *item);
    void setActiveItem(Q3CanvasItem *item);
    void showNewItem(Q3CanvasItem *item);

    Q3CanvasItem *pendingItem;
    Q3CanvasItem *activeItem;
    QPoint lastPos;
    int minZ;
    int maxZ;

    QAction *cutAct;
    QAction *copyAct;
    QAction *pasteAct;
    QAction *deleteAct;
    QAction *propertiesAct;
    QAction *addBoxAct;
    QAction *addLineAct;
    QAction *bringToFrontAct;
    QAction *sendToBackAct;
};

class DiagramBox : public Q3CanvasRectangle
{
public:
    enum { RTTI = 1001 };

    DiagramBox(Q3Canvas *canvas);

```

```

    DiagramBox(Q3Canvas *canvas, long x, long y);
    ~DiagramBox();

    void setText(const QString &newText);
    QString text() const { return str; }
    void drawShape(QPainter &painter);
    QRect boundingRect() const;
    int rtti() const { return RTTI; }

private:
    QString str;
};

class DiagramLine : public Q3CanvasLine
{
public:
    enum { RTTI = 1002 };

    DiagramLine(Q3Canvas *canvas);
    ~DiagramLine();

    QPoint offset() const { return QPoint((int)x(), (int)y()); }
    void drawShape(QPainter &painter);
    Q3PointArray areaPoints() const;
    int rtti() const { return RTTI; }
};

#endif

```

propertiesdialog.ui.h

```

#include <qcolordialog.h>

#include "cadview.h"

void PropertiesDialog::chooseLineColor()
{
    QColor oldColor = lineColorSample->paletteBackgroundColor();
    QColor newColor = QColorDialog::getColor(oldColor, this);
    if (newColor.isValid())
        lineColorSample->setPaletteBackgroundColor(newColor);
}

void PropertiesDialog::chooseFillColor()
{
    QColor oldColor = fillColorSample->paletteBackgroundColor();
    QColor newColor = QColorDialog::getColor(oldColor, this);
    if (newColor.isValid())
        fillColorSample->setPaletteBackgroundColor(newColor);
}

void PropertiesDialog::exec(Q3CanvasItem *item)
{
    if (item->rtti() == DiagramBox::RTTI) {
        DiagramBox *box = (DiagramBox *)item;
        QRect rect = box->rect();
        setCaption(tr("Properties for Box"));
    }
}

```



```

xSpinBox->setValue(rect.x());
ySpinBox->setValue(rect.y());
widthSpinBox->setValue(rect.width());
heightSpinBox->setValue(rect.height());
lineGeometryGroupBox->hide();
textLineEdit->setText(box->text());
lineColorSample->setPaletteBackgroundColor(
    box->pen().color());
fillColorSample->setPaletteBackgroundColor(
    box->brush().color());
} else if (item->rtti() == DiagramLine::RTTI) {
DiagramLine *line = (DiagramLine *)item;
QPoint start = line->startPoint() + line->offset();
QPoint end = line->endPoint() + line->offset();
setCaption(tr("Properties for Line"));

rectGeometryGroupBox->hide();
x1SpinBox->setValue(start.x());
y1SpinBox->setValue(start.y());
x2SpinBox->setValue(end.x());
y2SpinBox->setValue(end.y());
textLabel->hide();
textLineEdit->hide();
lineColorSample->setPaletteBackgroundColor(
    line->pen().color());
fillColorLabel->hide();
fillColorSample->hide();
fillColorButton->hide();
}

if (QDialog::exec()) {
if (item->rtti() == DiagramBox::RTTI) {
DiagramBox *box = (DiagramBox *)item;
box->move(xSpinBox->value(), ySpinBox->value());
box->setSize(widthSpinBox->value(),
    heightSpinBox->value());
box->setText(textLineEdit->text());
box->setPen(lineColorSample->paletteBackgroundColor());
box->setBrush(fillColorSample->paletteBackgroundColor());
} else if (item->rtti() == DiagramLine::RTTI) {
DiagramLine *line = (DiagramLine *)item;
line->move(0, 0);
line->setPoints(x1SpinBox->value(), y1SpinBox->value(),
    x2SpinBox->value(), y2SpinBox->value());
line->setPen(lineColorSample->paletteBackgroundColor());
line->setBrush(fillColorSample->paletteBackgroundColor());
}
item->canvas()->update();
}
}

```

mainwindow.xaml

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xml:lang="de-DE"
    x:Class="ResizingApp.MainWindow"
    x:Name="FirstWindow"
    Title="Softwareprototyp"
    Width="640" Height="480" Loaded="GetAdornerLayer"
    KeyDown="CaptureKey" KeyUp="StopCaptureKey" MouseMove="DragObject"
    Icon="kllckety.png">

    <Grid x:Name="LayoutRoot">
        <Rectangle Stroke="#FF000000" HorizontalAlignment="Left"
Margin="300,90,0,0" x:Name="FirstObject" VerticalAlignment="Top"
Width="40" Height="300" MouseLeftButtonDown="CaptureLeftButton"
MouseLeftButtonUp="StopCaptureLeftButton" Cursor="SizeAll">
            <Rectangle.Fill>
                <LinearGradientBrush EndPoint="1.155,-0.04"
StartPoint="0.01,0.86">
                    <GradientStop Color="#FF969EF0" Offset="0"/>
                    <GradientStop Color="#FF4C4BE2" Offset="1"/>
                </LinearGradientBrush>
            </Rectangle.Fill>
        </Rectangle>
    </Grid>
</Window>
```

mainwindow.xaml.cs

```
using System;
using System.IO;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Navigation;
using System.Windows.Documents;
using System.Windows.Input;

namespace ResizingApp
{
    public partial class MainWindow
    {
        AdornerLayer adornerLayer;

        public MainWindow()
        {
            this.InitializeComponent();
        }

        private void GetAdornerLayer(object sender,
System.Windows.RoutedEventArgs e)
        {
```

```

    adorningLayer = AdorningLayer.GetAdorningLayer(LayoutRoot);
}

private UIElement activeElement = null;
private Adorning activeAdorning = null;

public static bool CtrlIsDown = false;
public static bool HasBeenDragged = false;

private void CaptureKey(object sender,
System.Windows.Input.KeyEventArgs e)
{
    if (e.Key == Key.LeftCtrl || e.Key == Key.RightCtrl)
        CtrlIsDown = true;
}

private void StopCaptureKey(object sender,
System.Windows.Input.KeyEventArgs e)
{
    if (e.Key == Key.LeftCtrl || e.Key == Key.RightCtrl)
        CtrlIsDown = false;
}

public static bool LeftMouseIsDown = false;
public static double MouseLeft = 0;
public static double MouseTop = 0;

private void CaptureLeftButton(object sender,
System.Windows.Input.MouseButtonEventArgs e)
{
    UIElement uie = sender as UIElement;
    MouseLeft = e.GetPosition(uie).X;
    MouseTop = e.GetPosition(uie).Y;
    LeftMouseIsDown = true;
}

private void StopCaptureLeftButton(object sender,
System.Windows.Input.MouseButtonEventArgs e)
{
    if (!HasBeenDragged)
    {
        UIElement uie = sender as UIElement;

        if (uie != null)
        {
            if (uie == activeElement && activeAdorning != null)
            {
                adorningLayer.Remove(activeAdorning);
                activeElement = null;
                activeAdorning = null;
            }
            else
            {
                activeAdorning = new ResizingAdorning(uie);
                adorningLayer.Add(activeAdorning);
                activeElement = uie;
            }
        }
    }
}

```

```

    }
    LeftMouseDown = false;
    HasBeenDragged = false;
}

private void DragObject(object sender,
System.Windows.Input.MouseEventArgs e)
{
    if (LeftMouseDown)
    {
        if (activeAdorner != null)
        {
            adornerLayer.Remove(activeAdorner);
            activeAdorner = null;
        }

        HasBeenDragged = true;

        double posXInWindow = e.GetPosition(LayoutRoot).X;
        double posYInWindow = e.GetPosition(LayoutRoot).Y;

        Thickness dicke = new Thickness(posXInWindow - MouseLeft,
posYInWindow - MouseTop, 0, 0);
        FirstObject.Margin = dicke;
    }
}
}
}
}

```

resizingadorner.cs

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;

namespace ResizingApp
{
    public class ResizingAdorner : Adorner
    {
        Thumb topLeft, topRight, bottomLeft, bottomRight, Top, Bottom, Left,
Right;

        VisualCollection visualChildren;

        public ResizingAdorner(UIElement adornedElement)
            : base(adornedElement)
        {
            visualChildren = new VisualCollection(this);

```

```

BuildAdornerCorner(ref topLeft, Cursors.SizeNWSE);
BuildAdornerCorner(ref topRight, Cursors.SizeNESW);
BuildAdornerCorner(ref bottomLeft, Cursors.SizeNESW);
BuildAdornerCorner(ref bottomRight, Cursors.SizeNWSE);
BuildAdornerCorner(ref Top, Cursors.SizeNS);
BuildAdornerCorner(ref Bottom, Cursors.SizeNS);
BuildAdornerCorner(ref Left, Cursors.SizeWE);
BuildAdornerCorner(ref Right, Cursors.SizeWE);

bottomLeft.DragDelta += new DragDeltaEventHandler
(HandleBottomLeft);
bottomRight.DragDelta += new DragDeltaEventHandler
(HandleBottomRight);
topLeft.DragDelta += new DragDeltaEventHandler(HandleTopLeft);
topRight.DragDelta += new DragDeltaEventHandler(HandleTopRight);
Top.DragDelta += new DragDeltaEventHandler(HandleTop);
Bottom.DragDelta += new DragDeltaEventHandler(HandleBottom);
Left.DragDelta += new DragDeltaEventHandler(HandleLeft);
Right.DragDelta += new DragDeltaEventHandler(HandleRight);
}

void HandleBottomRight(object sender, DragDeltaEventArgs args)
{
    FrameworkElement adornedElement = this.AdornedElement as
FrameworkElement;
    Thumb hitThumb = sender as Thumb;

    EnforceSize(adornedElement);

    Rectangle rec = AdornedElement as Rectangle;

    rec.Height = Math.Max(rec.Height + args.HorizontalChange,
hitThumb.DesiredSize.Width);
    rec.Width = Math.Max(rec.Width + args.HorizontalChange,
hitThumb.DesiredSize.Width);
}

void HandleBottomLeft(object sender, DragDeltaEventArgs args)
{
    FrameworkElement adornedElement = AdornedElement as
FrameworkElement;
    Thumb hitThumb = sender as Thumb;

    if (adornedElement == null || hitThumb == null) return;

    EnforceSize(adornedElement);

    Rectangle rec = AdornedElement as Rectangle;
    Thickness dicke = new Thickness(rec.Margin.Left +
args.HorizontalChange, rec.Margin.Top, 0, 0);
    adornedElement.Margin = dicke;

    rec.Width = Math.Max(rec.Width - args.HorizontalChange,
hitThumb.DesiredSize.Width);
    rec.Height = Math.Max(rec.Height - args.HorizontalChange,
hitThumb.DesiredSize.Height);
}

void HandleTopRight(object sender, DragDeltaEventArgs args)

```

```

    {
        FrameworkElement adornedElement = this.AdornedElement as
FrameworkElement;
        Thumb hitThumb = sender as Thumb;

        if (adornedElement == null || hitThumb == null) return;
        FrameworkElement parentElement = adornedElement.Parent as
FrameworkElement;

        EnforceSize(adornedElement);

        Rectangle rec = AdornedElement as Rectangle;
        Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top +
args.VerticalChange, 0, 0);
        adornedElement.Margin = dicke;

        rec.Width = Math.Max(rec.Width - args.VerticalChange,
hitThumb.DesiredSize.Width);
        rec.Height = Math.Max(rec.Height - args.VerticalChange,
hitThumb.DesiredSize.Height);
    }

    void HandleTopLeft(object sender, DragDeltaEventArgs args)
    {
        FrameworkElement adornedElement = AdornedElement as
FrameworkElement;
        Thumb hitThumb = sender as Thumb;

        if (adornedElement == null || hitThumb == null) return;

        EnforceSize(adornedElement);

        Rectangle rec = AdornedElement as Rectangle;

        Thickness dicke = new Thickness(rec.Margin.Left +
args.HorizontalChange, rec.Margin.Top + args.VerticalChange, 0, 0);
        adornedElement.Margin = dicke;
        rec.Width = Math.Max(rec.Width - args.HorizontalChange,
hitThumb.DesiredSize.Width);
        rec.Height = Math.Max(rec.Height - args.HorizontalChange ,
hitThumb.DesiredSize.Height);
    }

    void HandleTop(object sender, DragDeltaEventArgs args)
    {
        FrameworkElement adornedElement = AdornedElement as
FrameworkElement;
        Thumb hitThumb = sender as Thumb;

        if (adornedElement == null || hitThumb == null) return;

        EnforceSize(adornedElement);

        Rectangle rec = AdornedElement as Rectangle;

        if (MainWindow.CtrlIsDown)
        {
            Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top +
args.VerticalChange/2, 0, 0);

```

```

        adornedElement.Margin = dicke;
        rec.Height = Math.Max(rec.Height - args.VerticalChange,
hitThumb.DesiredSize.Height);
    }
    else
    {
        Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top +
args.VerticalChange, 0, 0);
        adornedElement.Margin = dicke;
        rec.Height = Math.Max(rec.Height - args.VerticalChange,
hitThumb.DesiredSize.Height);
    }
}

void HandleBottom(object sender, DragDeltaEventArgs args)
{
    FrameworkElement adornedElement = AdornedElement as
FrameworkElement;
    Thumb hitThumb = sender as Thumb;

    if (adornedElement == null || hitThumb == null) return;

    EnforceSize(adornedElement);

    Rectangle rec = AdornedElement as Rectangle;
    Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top, 0,
0);
    adornedElement.Margin = dicke;

    rec.Height = Math.Max(rec.Height + args.VerticalChange,
hitThumb.DesiredSize.Height);
}

void HandleRight(object sender, DragDeltaEventArgs args)
{
    FrameworkElement adornedElement = this.AdornedElement as
FrameworkElement;
    Thumb hitThumb = sender as Thumb;

    if (adornedElement == null || hitThumb == null) return;
    FrameworkElement parentElement = adornedElement.Parent as
FrameworkElement;

    EnforceSize(adornedElement);

    Rectangle rec = AdornedElement as Rectangle;
    Thickness dicke = new Thickness(rec.Margin.Left, rec.Margin.Top, 0,
0);
    adornedElement.Margin = dicke;

    rec.Width = Math.Max(rec.Width + args.HorizontalChange,
hitThumb.DesiredSize.Width);
}

void HandleLeft(object sender, DragDeltaEventArgs args)
{
    FrameworkElement adornedElement = AdornedElement as
FrameworkElement;

```

```

    Thumb hitThumb = sender as Thumb;

    if (adornedElement == null || hitThumb == null) return;

    EnforceSize(adornedElement);

    Rectangle rec = AdornedElement as Rectangle;
    Thickness dicke = new Thickness(rec.Margin.Left +
args.HorizontalChange, rec.Margin.Top, 0, 0);
    adornedElement.Margin = dicke;

    rec.Width = Math.Max(rec.Width - args.HorizontalChange,
hitThumb.DesiredSize.Width);

}

protected override Size ArrangeOverride(Size finalSize)
{
    double adornerWidth = AdornedElement.DesiredSize.Width;
    double adornerHeight = AdornedElement.DesiredSize.Height;
    double adornerSize = 5;

    Rectangle rec = AdornedElement as Rectangle;

    adornerWidth = adornerWidth - rec.Margin.Left;
    adornerHeight = adornerHeight - rec.Margin.Top;

    topLeft.Arrange(new Rect(-adornerWidth / 2 + adornerSize,
-adornerHeight / 2 + adornerSize, adornerWidth, adornerHeight));
    topRight.Arrange(new Rect(adornerWidth / 2 - adornerSize,
-adornerHeight / 2 + adornerSize, adornerWidth, adornerHeight));
    bottomLeft.Arrange(new Rect(-adornerWidth / 2 + adornerSize,
adornerHeight / 2 - adornerSize, adornerWidth, adornerHeight));
    bottomRight.Arrange(new Rect(adornerWidth / 2 - adornerSize,
adornerHeight / 2 - adornerSize, adornerWidth, adornerHeight));
    Top.Arrange(new Rect(0, -adornerHeight / 2 + adornerSize,
adornerWidth, adornerHeight));
    Bottom.Arrange(new Rect(0, adornerHeight / 2 - adornerSize,
adornerWidth, adornerHeight));
    Left.Arrange(new Rect(-adornerWidth / 2 + adornerSize, 0,
adornerWidth, adornerHeight));
    Right.Arrange(new Rect(adornerWidth / 2 - adornerSize, 0,
adornerWidth, adornerHeight));

    return finalSize;
}

void BuildAdornerCorner(ref Thumb cornerThumb, Cursor
customizedCursor)
{
    if (cornerThumb != null) return;

    cornerThumb = new Thumb();

    cornerThumb.Cursor = customizedCursor;
    cornerThumb.Height = cornerThumb.Width = 10;
    cornerThumb.Opacity = 0.70;
    cornerThumb.Background = new SolidColorBrush
(Colors.CornflowerBlue);
}

```



```

        visualChildren.Add(cornerThumb);
    }

    void EnforceSize(FrameworkElement adornedElement)
    {
        if (adornedElement.Width.Equals(Double.NaN))
            adornedElement.Width = adornedElement.DesiredSize.Width;

        if (adornedElement.Height.Equals(Double.NaN))
            adornedElement.Height = adornedElement.DesiredSize.Height;

        FrameworkElement parent = adornedElement.Parent as
FrameworkElement;
        if (parent != null)
        {
            adornedElement.MaxHeight = parent.ActualHeight;
            adornedElement.MaxWidth = parent.ActualWidth;
        }
    }

    protected override int VisualChildrenCount { get { return
visualChildren.Count; } }
    protected override Visual GetVisualChild(int index) { return
visualChildren[index]; }
    }
}

```

Software-Prototyp IV - Erweiterung

mainwindow.xaml

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xml:lang="de-DE"
    x:Class="ResizingApp.MainWindow"
    x:Name="FirstWindow"
    Title="Direktmanipulation"
    Width="640" Height="480" Loaded="GetAdornerLayer"
    KeyDown="CaptureKey" KeyUp="StopCaptureKey" Icon="kllckety.png">
    <Grid x:Name="LayoutRoot"></Grid>
</Window>

```

mainwindow.xaml.cs

```

using System;
using System.IO;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Navigation;
using System.Windows.Documents;

```

```

using System.Windows.Input;
using System.Windows.Shapes;

namespace ResizingApp
{
    public partial class MainWindow
    {
        AdornerLayer adornerLayer;

        public MainWindow()
        {
            this.InitializeComponent();
            this.Loaded += new RoutedEventHandler(MainWindow_Loaded);
        }

        private void MainWindow_Loaded(object sender, RoutedEventArgs e)
        {
            CreateRectangle(300, 80, "Stab", 40, 300);
        }

        private Rectangle CreateRectangle(double x, double y, string
name, double width, double height)
        {
            Rectangle rect = new Rectangle();
            rect.Stroke = new SolidColorBrush(Colors.Black);
            rect.Fill = new SolidColorBrush(Colors.LightGray);
            rect.HorizontalAlignment = HorizontalAlignment.Left;
            rect.Margin = new Thickness(x, y, 0, 0);
            rect.Name = name;
            rect.VerticalAlignment = VerticalAlignment.Top;
            rect.Width = width;
            rect.Height = height;
            rect.MouseLeftButtonDown += new MouseButtonEventHandler
(CaptureLeftButton);
            rect.MouseLeftButtonUp += new MouseButtonEventHandler
(StopCaptureLeftButton);
            rect.MouseMove += new MouseEventHandler(DragObject);

            LayoutRoot.Children.Add(rect);

            return rect;
        }

        private void GetAdornerLayer(object sender,
System.Windows.RoutedEventArgs e)
        {
            adornerLayer = AdornerLayer.GetAdornerLayer(LayoutRoot);
        }

        private UIElement activeElement = null;
        private Adorner activeAdorner = null;

        public static bool CtrlIsDown = false;
        public static bool DontCheckCtrl = false;
        public static bool HasBeenDragged = false;

        private void CaptureKey(object sender,
System.Windows.Input.KeyEventArgs e)

```

```

    {
        if (e.Key == Key.LeftCtrl)
            CtrlIsDown = true;
    }

    private void StopCaptureKey(object sender,
System.Windows.Input.KeyEventArgs e)
    {
        if (e.Key == Key.LeftCtrl)
        {
            CtrlIsDown = false;
            RectangleJustCreated = false;
        }
    }

    public static bool LeftMouseIsDown = false;
    public static double MouseLeft = 0;
    public static double MouseTop = 0;

    private void CaptureLeftButton(object sender,
System.Windows.Input.MouseButtonEventArgs e)
    {
        UIElement uie = sender as UIElement;
        MouseLeft = e.GetPosition(uie).X;
        MouseTop = e.GetPosition(uie).Y;
        LeftMouseIsDown = true;
    }

    private void StopCaptureLeftButton(object sender,
System.Windows.Input.MouseButtonEventArgs e)
    {
        if (!HasBeenDragged)
        {
            UIElement uie = sender as UIElement;

            if (uie != null)
            {
                if (uie == activeElement && activeAdorner != null)
                {
                    adornerLayer.Remove(activeAdorner);
                    activeElement = null;
                    activeAdorner = null;
                }

                else
                {
                    activeAdorner = new ResizingAdorner(uie);
                    adornerLayer.Add(activeAdorner);
                    activeElement = uie;
                }
            }
        }

        LeftMouseIsDown = false;
        HasBeenDragged = false;
    }

    private static bool RectangleJustCreated = false;

```

```

        private void DragObject(object sender,
System.Windows.Input.MouseEventArgs e)
        {
            Rectangle clickedRect = sender as Rectangle;
            if (LeftMouseDown && clickedRect != null)
            {
                if (activeAdorner != null)
                {
                    adornerLayer.Remove(activeAdorner);
                    activeAdorner = null;
                }

                if (CtrlIsDown && !RectangleJustCreated)
                {
                    Rectangle rect = CreateRectangle
(clickedRect.Margin.Left, clickedRect.Margin.Top, "CopyOf" +
clickedRect.Name, clickedRect.Width, clickedRect.Height);
                    clickedRect = rect;
                    RectangleJustCreated = true;
                }

                HasBeenDragged = true;

                double posXInWindow = e.GetPosition(LayoutRoot).X;
                double posYInWindow = e.GetPosition(LayoutRoot).Y;

                Thickness dicke = new Thickness(posXInWindow - MouseLeft,
posYInWindow - MouseTop, 0, 0);
                clickedRect.Margin = dicke;
            }
        }
    }
}

```