Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

STUDIENARBEIT

# Generische Visualisierungsdienste am Beispiel von StarView und Tycoon

April 1996

Ilona Hachmann

**Betreuer:**
Prof. Dr. Joachim W. Schmidt
Gerald Schröder

# Inhaltsverzeichnis

**1. Einleitung** **1**
1.1 Zielsetzung und Aufbau der Arbeit . . . . . 2

**2. Entwicklungswerkzeuge für graphische Benutzerschnittstellen** **3**
2.1 Fenstersysteme . . . . . 4
2.2 Resource-Editoren . . . . . 4
2.3 Klassenbibliotheken . . . . . 6
2.4 Systemübergreifende Bibliotheken . . . . . 8
2.5 Klassifizierung . . . . . 8

**3. StarView und Tycoon** **11**
3.1 Grundlegende GUI-Elemente . . . . . 11
3.2 Die StarView Klassenhierarchie . . . . . 12
3.2.1 Klassenübersicht nach Kategorie . . . . . 12
3.3 Grundelemente einer StarView-Anwendung . . . . . 15

**4. Entwicklung graphischer Benutzeroberflächen mit Tycoon und StarView** **19**
4.1 Der implizite Weg . . . . . 19
4.1.1 Aufbau der Editoren . . . . . 20
4.1.2 Beispiel eines Basiseditors . . . . . 20
4.1.3 Aufbau der Editorbibliothek . . . . . 22
4.2 Der explizite Weg . . . . . 22
4.2.1 HelloWorld . . . . . 23
4.2.2 Painter . . . . . 24
4.2.3 PainterWithAboutBox . . . . . 30
4.3 Der interaktive Weg . . . . . 34
4.4 Vor- und Nachteile der Wege . . . . . 41

**5. Zusammenfassung und Ausblick** **43**
5.1 Erfahrungen mit StarView . . . . . 43
5.2 Erfahrungen mit dem Design Editor . . . . . 44
5.3 Ausblick . . . . . 44
5.4 Zusammenfassung . . . . . 45

**A. Abbildung von C++-Klassen auf TL Module** **46**

**B. Die Beispiele "HelloWorld", "Painter", "PainterWithAboutBox", "PainterWithResources"** **50**

B.1 Das Beispiel "HelloWorld" . . . . . 50
B.2 Das Beispiel "Painter" . . . . . 52
B.3 Das Beispiel "PainterWithAboutBox" . . . . . 57
B.4 Das Beispiel "PainterWithResources" . . . . . 63

**Literaturverzeichnis** **73**

# Abbildungsverzeichnis


2.1 Der Design Editor . . . . . 5
2.2 Ausschnitt aus der StarView Klassenhierarchie . . . . . 7
2.3 Klassifizierungsschema für Entwicklungswerkzeuge . . . . . 9

3.1 Ein erstes StarView Beispiel . . . . . 16
3.2 Die Ereignisverarbeitung (*event-loop*) . . . . . 18

4.1 Der implizite Weg . . . . . 20
4.2 Editor zum Anzeigen von Zeichenketten . . . . . 21
4.3 Struktur der Editorbibliothek . . . . . 22
4.4 Der explizite Weg . . . . . 23
4.5 Das Hauptfenster des Beispiels "Painter" . . . . . 24
4.6 Die Menüleiste des Beispiels "Painter" . . . . . 25
4.7 Das Popupmenü *Color* . . . . . 25
4.8 Die Messagebox zum Verlassen von "Painter" . . . . . 30
4.9 Das Hauptfenster des Beispiels "PainterWithAboutBox" . . . . . 31
4.10 Die Menüleiste des Beispiels "PainterWithAboutBox" . . . . . 31
4.11 Das Popupmenü *File* des Beispiels "PainterWithAboutBox" . . . . . 32
4.12 Die *AboutBox* des Beispiels "PainterWithAboutBox" . . . . . 32
4.13 Der Design Editor . . . . . 35
4.14 Bearbeitung des Menüeintrags *File* mit dem Design Editor . . . . . 37
4.15 Liste der Schlüsselwörter der *AboutBox* . . . . . 39
4.16 Bearbeitung der *AboutBox* mit dem Design Editor . . . . . 40

# 1. Einleitung

Den Vorgang der Sichtbarmachung von Materie, Energie, Informationen oder Prozessen bezeichnet man als Visualisierung. [Kröm 92] beschränkt diesen Vorgang in der Informatik auf die Visualisierung von Daten, den direkten Repräsentanten von Informationen. Damit Daten für den Menschen visuell wahrnehmbar werden, müssen sie in eine geeignete Darstellung gewandelt werden. Die Herkunft der Daten ist dabei unerheblich. Sie könnten z.B. interaktiv erfaßt oder durch Simulation oder Messung gewonnen worden sein. Das Resultat einer Visualisierung ist auf jeden Fall eine optische Erscheinung, die auf den Betrachter einen visuellen Reiz ausübt.
Visualisierung[1] ist die wichtigste Möglichkeit zur Präsentation von Daten. Das Auge ist das Hauptsinnesorgan des Menschen, mit ihm nimmt der Mensch den größten Teil an Informationen auf, die von außen auf ihn einwirken [Fran 87].
Für Blinde ist eine Ausgabe der Daten auf ein spezielles Tastfeld möglich. Für sie ist die "Materialisierung" die wichtigste Möglichkeit zur Präsentation von Daten. Andere Darstellungsmöglichkeiten, wie z.B. Audio, sind für komplexe Zusammenhänge nicht geeignet.
Die Kommunikation des Menschen mit dem Rechner über diese Daten erfolgt textuell oder mit graphischen Benutzerschnittstellen (*graphical user interfaces*, GUIs). GUIs sollen die Arbeit mit dem Computer erleichtern und komplexe Anwendungsprogramme intuitiv bedienbar machen. Aus der Vielzahl von Einsatzbereichen ergeben sich die unterschiedlichsten Anforderungen an ein GUI. GUIs sind gerade für datenintensive Anwendungen interessant [Müßig 94]. Die Funktionalität des GUIs muß dahin ausgerichtet sein, in großen und komplexen Datenbeständen zu navigieren und dem Benutzer seine definierte Auswahl anzuzeigen oder zu verändern. Die Daten können einfache Zahlen, Zeichenketten oder sogar komplexe Multimediaobjekte sein.

Die Anforderungen an die Entwicklung einer graphischen Benutzerschnittstelle sind sehr hoch. Vom Programmierer werden Kenntnisse im Umgang mit dem verwendeten Werkzeug zur Erstellung der Benutzerschnittstelle (siehe Kapitel 2) sowie Kenntnisse von Stilrichtlinien (*style guides*)[2] verlangt. Die Implementation ist sehr resourcenintensiv und beansprucht viel Zeit [Fähnrich et al. 92].
Der Aufwand bei der Erstellung einer graphischen Benutzeroberfläche steht nicht im po-

[1] Mit Visualisierung ist hier neben Graphikdarstellung auch Textdarstellung gemeint.
[2] In Stilrichtlinien wird die Gestaltung von Benutzeroberflächen vorgegeben.

sitiven Verhältnis zu der erhofften Vereinfachung bei der Bedienung der Applikation. Die Ursachen sind vielfältig. So schätzen Entwickler den Anteil der graphischen Benutzeroberfläche am Programmkode im Moment auf etwa 60%, Tendenz steigend [Wegener 95]. Der iterative Entwicklungsprozeß, hohe Anforderungen an Geschwindigkeit und Robustheit sind weitere Ursachen [Müßig 94]. Dadurch verbreiten und entwickeln sich graphische Benutzerschnittstellen nicht schnell. Um dem entgegenzuwirkenn werden vermehrt verschiedene Techniken zur Erstellung von graphischen Benutzeroberflächen angeboten.

## 1.1 Zielsetzung und Aufbau der Arbeit

Diese Arbeit soll Tycoon-Anwendungsprogrammierern als Anleitung dafür dienen, wie sie ihre Anwendung mit einer graphischen Benutzerschnittstelle versehen können. Dazu wird in Kapitel 2 eine Übersicht über mögliche Entwicklungswerkzeuge gegeben. In Kapitel 3 wird das Produkt StarView vorgestellt, eine Kombination aus zwei Entwicklungswerkzeugen. Eine Einführung in die Oberflächenprogrammierung mit Tycoon unter Verwendung der StarView Klassenbibliothek mit Programmierbeispielen bietet Kapitel 4. Dort wird ebenfalls die Verwendung von Resourcen und die Verbindung zum StarView Design Editor erläutert. Die Erfahrungen mit StarView und dem Design Editor sowie ein Ausblick auf mögliche Anschlußarbeiten sind Bestandteile des abschließenden Kapitels 5.

# 2. Entwicklungswerkzeuge für graphische Benutzerschnittstellen

Bei vielen graphischen Benutzeroberflächen ist der Kode des Anwendungsprogramms mit dem der Benutzeroberfläche vermischt [Wegener 95]. Aus diesem Programm später die oberflächenspezifischen Funktionen herauszufinden, ist fast unmöglich. Bei solchen Programmen bestimmt die Anwendung das Aussehen der Oberfläche.
Durch die separate Entwicklung der Oberfläche erreicht man eine Trennung vom Programmkode der Anwendung. Nicht nur die Wartung der beiden Teile ist einfacher, sondern auch kürzere Entwicklungszeiten und einfacheres Prototyping sind möglich [Müßig 94].
In [Wegener 95] wird gefordert, daß diese beiden Komponenten möglichst nicht von derselben Person programmiert werden. Ein Experte mit ergonomischen Kenntnissen sollte das Oberflächen-Design übernehmen (*User-Interface-Designer*). Unabhängig von der Anwendung kann er ergonomisch günstige Lösungen einer Oberfläche entwerfen.
In [SchwPin 95] wird die Zusammenarbeit von Programmierern und Designern beschrieben. Zusammenfassend sind die Anforderungen an ein Entwicklungswerkzeug:

- Trennung zwischen Anwendung und Benutzerschnittstelle
- einfache Bedienung
- wenig Programmkode
- fenstersystemunabhängig
- plattformunabhängig
- flexibel

Mit Resource-Editoren, z.B. dem Design Editor von StarView, kann aus einer Fülle von vordefinierten Interaktionselementen zunächst ein statisches Abbild der Benutzerschnittstelle erstellt werden, um diese an die Anwendung zu binden. Für diese Aufgabe werden unterschiedliche Methoden respektive Werkzeuge zur Vefügung gestellt. Im Folgenden werden anhand von StarView Klassenbibliotheken, systemübergreifende Klassenbibliotheken sowie

Resource-Editoren vorgestellt werden. Das Kapitel schließt mit einer Klassifizierung der Ansätze.

## 2.1 Fenstersysteme

Fenstersysteme (*window system*) bilden die Grundlage der Erstellung von direktmanipulativen Benutzerschnittstellen [Müßig 94]. Mit einer vom Fenstersystem gestellten Bibliothek mit Basisfunktionen greift man auf das Fenstersystem zu. Weil Fenstersysteme nur eine Basisfunktionalität anbieten, ist der Entwurf einer Benutzerschnittstelle auf dieser Ebene sehr mühsam. Komplexere Oberflächen müssen aus mehreren Funktionsaufrufen zusammengesetzt werden.
Am Arbeitsbereich *Datenbanken und Informationsysteme* (DBIS) sind u.a. folgende Fenstersysteme mit den dazugehörigen Stilrichtlinien und Bibliotheken vorhanden: X11 mit OSF/Motif und der Xlib, OpenWindows mit OPEN LOOK und der Xlib für Unix-Systeme, der Finder mit den UI Guidelines und QuickDraw für den Macintosh, sowie Microsoft Windows mit dem Design Guide und Win32 für den PC. Die Kombination Tycoon/StarView läuft auf Solaris 2.x bzw. SunOS 5.x, sowie Macintosh und Windows NT.

## 2.2 Resource-Editoren

Bei Resource-Editoren hat der Oberflächen-Entwickler die Möglichkeit, interaktiv eine Benutzerschnittstelle zu erstellen. Es steht ein Werkzeugkasten zur Verfügung, aus dem per *Drag & Drop* Interaktionselemente ausgewählt und plaziert werden können. Während des Entwurfs ist das spätere Aussehen bereits zu erkennen und kann gegebenenfalls auch sofort wieder geändert werden (WYSIWYG-Prinzip – *What You See Is What You Get*). Resource-Editoren erzeugen aus dem graphischen Entwurf eine binäre Datei, die Resource-Datei.
StarView bietet den Design Editor (siehe Abbildung 2.1) an. Dieser ist selbst eine StarView-Applikation.

Resource-Dateien können manuell erstellt oder vom Design Editor automatisch generiert werden. In diesen Dateien ist eine Beschreibung des Aussehens der Benutzeroberfläche abgelegt, z.B. Aufbau der Menüs, Art der Einträge, standardmäßig gesetzte Eigenschaften, Größe und Position von Dialogboxen, Tastenkürzel etc. Ein Vorteil von Resource-Dateien liegt darin, daß sie für mehrere Anwendungen benutzt werden können. Resource-Dateien werden getrennt von der Anwendung übersetzt; das bedeutet, daß bei Modifikationen im Programm lediglich die betreffenden Resource-Dateien zu ändern und neu zu übersetzen sind. Der folgende Programmkode zeigt ein Beispiel für eine Resource-Datei von StarView. Es wird eine Dialogbox mit einem Knopf "Ok" und einem Text "StarView" beschrieben.

```
/* StarView resource file */
#pragma CHARSET_ANSI
#include "painterWithResourcesAbout.hrc"
```

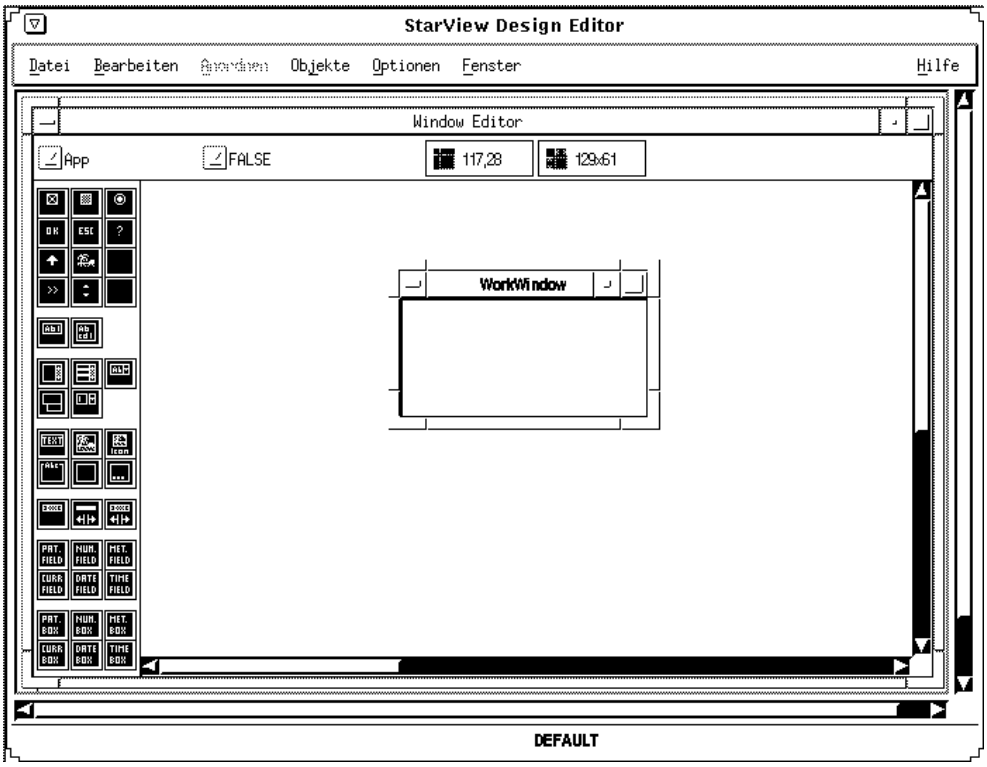


Abbildung 2.1: Der Design Editor

```
ModalDialog RID_ABOUTBOX {
        Hide = TRUE;
        Pos = MAP_APPFONT( 45, 29 );
        Size = MAP_APPFONT( 156, 126 );
        Text = "Paint";
        Moveable = TRUE;
        OKButton RID_OK {
                        Pos = MAP_APPFONT( 54, 105 );
                        Size = MAP_APPFONT( 47, 10 );
                        Text = "~OK";
                        TabStop = TRUE;
                        DefButton = TRUE;
                };
        FixedText RID_TEXT {
                        Pos = MAP_APPFONT( 56, 73 );
                        Size = MAP_APPFONT( 40, 18 );
                        Text = "StarView";
```

```
            Left = TRUE;
        };
};
```

## 2.3 Klassenbibliotheken

Klassen sind rein statische Beschreibungen einer Menge von möglichen Objekten [Meyer 88]. In einer Klasse werden nicht individuelle Objekte beschrieben, sondern gemeinsame Eigenschaften (Merkmale und Funktionen) von Objekten. Eine Klasse beschreibt die Struktur einer (eventuell unendlichen) Menge von Objekten. Jedes dieser Objekte wird Exemplar (*instance*) dieser Klasse genannt. Klasse ist ein Begriff der Übersetzungszeit (*compile-time*), während Objekte nur zur Laufzeit (*run-time*) existieren.
Zwischen Klassen können Vererbungsbeziehungen existieren [MoSch 92]. Subklassen werden aus Superklassen abgeleitet. Subklassen erben alle Eigenschaften der Superklasse. Die Subklassen können in den Eigenschaften modifiziert bzw. erweitert werden. Werden aus einer Subklasse weitere Subklassen abgeleitet entsteht eine Klassenhierarchie. Der Vorteil von Klassenbibliotheken, viel Funktionalität zu bieten, kann auch zum Nachteil werden [Borland 94]. Mit wachsendem Funktionsumfang steigt die Komplexität der Bibliotheken und damit der Einarbeitungsaufwand für Entwickler. Abhilfe schaffen hier Klassenhierarchien. Sie sollen nicht nur für die Beibehaltung des *look and feel* des jeweils zugrundeliegenden Fenstersystems sorgen, sondern auch dem Entwickler eine einfache Handhabung der Bibliotheken ermöglichen.
Die Abbildung 2.2 zeigt ein Beispiel für die Klassenhierarchie einer Klassenbibliothek, die StarView Klassenbibliothek.

Am AB DBIS wurde bis vor kurzem das NeWS Toolkit für OPEN LOOK benutzt. Das NeWS Toolkit wurde mit Erscheinen von Solaris 2.3 nicht mehr von Sun Microsystems unterstützt. Da das Tycoon-System inzwischen auch auf andere Plattformen portiert wurde, hat man sich für den Einsatz der systemunabhängigen Klassenbibliothek StarView entschieden.

Klassenbibliotheken folgen den Konzepten der objektorientierten Programmierung [Markus 92]. In einer objektorientierten Sprache (z.B. C++ oder Smalltalk) werden spezielle Klassen zur Verfügung gestellt. In diesen speziellen Klassen sind Methoden zusammengefaßt, die die Programmierung von graphischen Benutzerschnittstellen unterstützen. Die Funktionalität dieser Klassen wäre sonst nur durch eine Vielzahl von Funktionsaufrufen der Fenstersystemschnittstelle zu erreichen. Diese Zusammenfassung der Funktionen verringert die Fehlerquote, da weniger Programmkode anfällt. Durch objektorientierte Ideen wie Vererbung, Klassen, Modularisierung werden Redundanzen vermieden sowie Datenkapselung und Wiederverwendbarkeit erreicht.
Klassen, die nicht ganz den Anforderungen entsprechen, können durch Definieren einer Subklasse modifiziert bzw. erweitert werden. Alle Eigenschaften der Superklasse, bis auf die veränderten, werden an die Subklasse vererbt. Sollen nur wenige bestimmte Eigenschaften

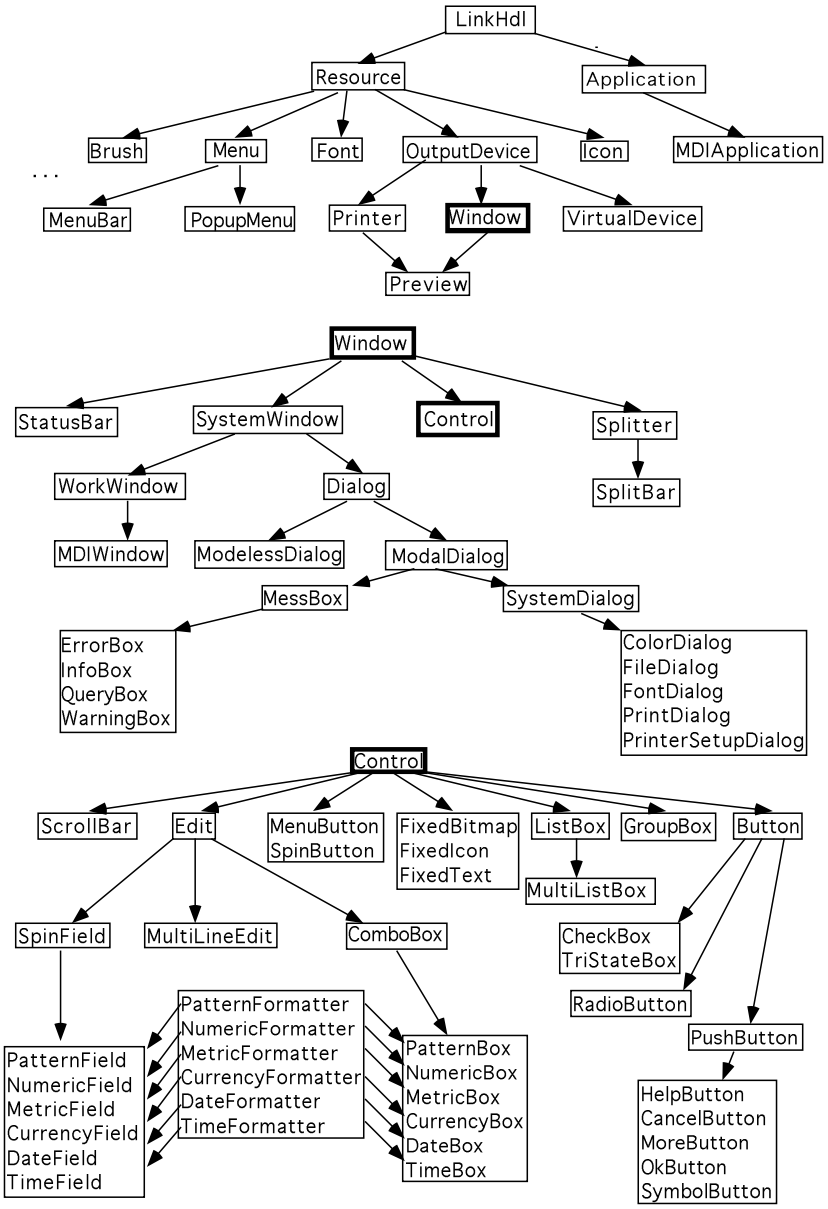


Abbildung 2.2: Ausschnitt aus der StarView Klassenhierarchie

verändert werden, so überschreibt man sie explizit. Der Vorteil ist automatische Konsistenz der anderen Eigenschaften über alle Instanzen hinweg.
Da bei der Erstellung von Anwendungen mit Bibliotheken der Programmkode von Hand erstellt wird, muß die Entwicklerdisziplin sehr groß sein, um portablen Kode zu erhalten [Isau 93b].

## 2.4 Systemübergreifende Bibliotheken

"Systemübergreifende Bibliotheken (*virtual toolkits*) bieten identische Programmierschnittstellen und eine identische Sprache für die Beschreibung von Benutzerschnittstellen für mehrere Systemplattformen an und ermöglichen so die einfache Portierung von Anwendungsprogrammen auf verschiedene Fenstersysteme" [Markus 92, S. 42].
Systemübergreifende Bibliotheken stehen zwischen dem Anwendungsprogramm und dem Fenstersystem. Alle Funktionsaufrufe der systemunabhängigen Programmierschnittstelle werden von der systemübergreifenden Bibliothek in die entsprechenden Funktionsaufrufe des Fenstersystems umgewandelt. So erreicht man Portabilität des Anwendungsprogramms.
Die Frage bei der Verschiedenartigkeit der von verschiedenen Fenstersystemen unterstützten Funktionen ist: Welche Funktionen soll eine Klassenbibliothek anbieten? Ein Ansatz ist, daß nur eine gemeinsame Menge der Funktionalität aller Fenstersysteme unterstützt wird (Schnittmengenansatz). Das hat zur Folge, daß das System mit der kleinsten Funktionalität alle anderen Systeme auf sich reduziert. Fehlende Komponenten müssen vom Programmierer entworfen werden.
Eine andere Möglichkeit ist, alle Funktionen und Elemente aller durch die Klassenbibliothek unterstützten Fenstersysteme anzubieten (Obermengenansatz). Funktionen, die ein System nicht aufweist, dafür aber bei einem anderen System verfügbar sind, werden durch die Bibliothek nachgebildet [Müßig 94].
Systemunabhängige Bibliotheken können auch Nachteile haben [Isau 93a]. Häufig wird die Plattformunabhängigkeit der Anwendung durch Abhängigkeit vom Hersteller der Klassenbibliothek erkauft. So sind Anwendungen, die mit den Produkten StarView oder XVT erstellt worden sind, nicht untereinander austauschbar.

## 2.5 Klassifizierung

Die Klassifizierung der Werkzeuge erfolgt nach einem Schema aus [Fähnrich et al. 92]. Dieses Schema (siehe Abbildung 2.3) zeigt, daß es noch mehr Entwicklungswerkzeuge gibt; diese sind bei StarView nicht vorhanden.
Auf der vertikalen Achse sind die Architekturschichten aufgetragen, die das Werkzeug unterstützt. Auf der horizontalen Achse ist der Abstraktionsgrad des Werkzeuges abgetragen. Er steigt von links nach rechts.

Die Anwendungsschichten gliedern sich in Präsentationsschicht, Dialogsteuerung und Anwendungslogik. In der Präsentationsschicht werden externe Darstellungen der Benutzerober-

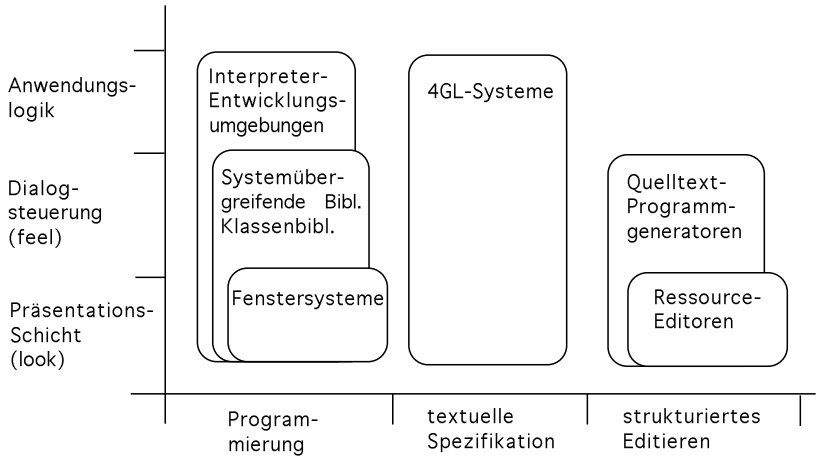


Abbildung 2.3: Klassifizierungsschema für Entwicklungswerkzeuge

fläche verwaltet, sowie die Interaktion mit dem Anwender ermöglicht. Die Dialogsteuerung erhält die Eingaben des Anwenders von der Präsentationsschicht. Die Eingaben werden interpretiert und eine entsprechende Aktion wird ausgeführt. Der Ablauf des Dialoges zwischen Anwender und Anwendung wird also definiert. Die Anwendungsschicht enthält die Objekte sowie die eigentlichen Funktionen der Anwendung, die die Objekte bearbeiten.

Programmierwerkzeuge wie Fenstersysteme bieten den niedrigsten Abstraktionsgrad. Der Entwickler benutzt sie von allgemeinen Programmiersprachen aus. Ähnlich ist es mit Klassenbibliotheken und systemübergreifenden Klassenbibliotheken. Aufgrund des großen Funktionsumfang sind diese Werkzeuge jedoch recht komplex. Bei unzureichender Dokumentation kann gerade der Anfänger vor der Komplexität und dem großen Funktionsumfang resignieren.
4GL-Systeme ermöglichen die textuelle Spezifikation von vollständigen Benutzerschnittstellen. In den speziellen Beschreibungssprachen sind Konstrukte für die Beschreibung von Schnittstellenkomponenten, (z.B. Fenstern, Menüs und Knöpfen, und ihren Darstellungsattributen, wie Position, Größe und Farbe), enthalten.
Strukturierte Editoren haben den höchsten Abstraktionsgrad in diesem Schema.
Über einen Schnittstellenkomponentenbaukasten ist die direktmanipulative Entwicklung, sowie die Attributierung der Komponenten einer graphischen Oberfläche möglich. Im Hinblick auf den zunehmenden Programmkode bei der Gestaltung von graphischen Oberflächen werden immer mehr Klassenbibliotheken mit *interface-buildern* verknüpft [Schwarz 94a].
Soll die graphische Oberfläche zusätzlich portabel sein, so ist eine systemübergreifende Klassenbibliothek zu verwenden.
Durch die Anbindung der systemübergreifenden Klassenbibliothek StarView an das Tycoon-System kann eine graphische Benutzerschnittstelle dynamisch erzeugt und verändert werden.

Mit dem Design Editor (einem Ressource Editor) von StarView können auf einfache Weise statische Teile einer Benutzerschnittstelle erzeugt werden.

# 3. StarView und Tycoon

StarView ist eine portable C++ Klassenbibliothek, d.h. die Entwicklung und Gestaltung der graphischen Oberfläche kann unabhängig vom jeweiligen Fenstersystem erfolgen. Graphische Benutzeroberflächen zeichnen sich aus durch das Erscheinungsbild und die Funktionalität der Fenster und Interaktionselemente (*look*) einerseits und das Verhalten des Systems auf Benutzerinteraktionen (*feel*) andererseits. Für die Beibehaltung des *look and feel* des jeweils zugrundeliegenden Fenstersystems sorgt die Klassenhierarchie der StarView C++ Bibliothek.

Im Abschnitt 2.4 über systemübergreifende Bibliotheken wurden der Schnittmengen- und Obermengenansatz erläutert. StarView stellt eine Mischform der beiden Ansätze dar [Geisler 95]. Der Schnittmengenansatz sorgt für eine hohe Portabilität, aber auch, wie bereits erwähnt, für eine große Einschränkung der Funktionalität. Der Obermengenansatz kann die Portabilität sehr schnell wieder einschränken. Man muß bei der Programmierung einer GUI-Anwendung mit einer systemübergreifenden Klassenbibliothek (hier StarView) also darauf achten, nur Funktionalität zu verwenden, die auf allen unterstützten Fenstersystemen vorhanden ist, oder von vornherein wissen, daß die Anwendung nur auf einem bestimmten System laufen soll. Ist das System bekannt, so kann gleich ein spezifisches Oberflächenentwicklungswerkzeug benutzen werden. Dieses Werkzeug bietet dann volle Funktionalität [Schwarz 94a].
Dieses Kapitel gibt einen Überblick über die in StarView vorhandenen Klassen und nimmt eine Zuordnung der Klassen zu Funktionsgruppen vor. Das Kapitel schließt mit einem ersten einführenden Beispiel.

## 3.1 Grundlegende GUI-Elemente

In diesem Abschnitt sollen die wichtigsten GUI-Elemente beschrieben werden. Diese Elemente finden sich als Klassen in der StarView-Klassenhierarchie wieder. Weitere, speziellere Elemente finden sich in den Beispielen.

### Fenster (*windows*)

Fenster sind in Anwendungen mit graphischen Oberflächen wesentliches Medium der Interaktion. "Sie dienen der Präsentation der Anwendungsdaten und der Entgegennahme der

Eingaben des Benutzers" [Geisler 95, S. 36]. Über spezielle Interaktionselemente im Fenster werden die Eingaben entgegengenommen. Das Fenster muß in einem bestimmtem Status sein (*focus*), damit es Eingaben, z.B. Tastatureingaben, empfangen kann.

#### Interaktionselemente (*controls*)

Interaktionselemente ermöglichen dem Benutzer die Kommunikation mit der Anwendung. Durch Maus oder Tastatur werden Schnittstellenelemente wie Knöpfe, Eingabefelder, Menüs, Rollbalken etc. manipuliert. Diesen Elementen kann eine Funktionalität zugeordnet werden, die dann bei entsprechender Aktion des Benutzers ausgeführt wird.

#### Dialoge

Dialoge fassen mehrere Interaktionselemente zusammen. Aus einer Anzahl von Elementen wählt der Benutzer die für ihn wichtigen aus, die den weiteren Verlauf des Programms bestimmen. Dialoge steuern, welche Eingaben zu welchem Fenster gehören und garantieren damit eine zusammenhängende Bearbeitung der Interaktionselemente. StarView unterscheidet zwischen *modalen* und *nichtmodalen* Dialogen. Modale Dialoge müssen, im Gegensatz zu nichtmodalen Dialogen, beendet werden, bevor in der Anwendung fortgefahren werden kann. Die Verwendung von modalen Dialogen kann die benutzergesteuerte Eingabe einschränken [HüSha 93].

#### Datenaustausch

Daten können zwischen mehreren Anwendungen transferiert werden. Dies unterstützen Verfahren wie Klemmbrett (*clipboard*) oder *drag & drop*. Das Klemmbrett ist eine Art Zwischenspeicher, in den per Benutzeraktionen 'Ausschneiden' (*cut*) und 'Kopieren' (*copy*) Daten gestellt werden. Mit der Benutzeraktion 'Einfügen' (*paste*) können Kopien von den im Klemmbrett befindlichen Daten angefordert werden. Sind die Daten mit der Maus selektierbar (z.B. Zeichenketten und Zahlen), so kann man sie mit der Maus aufnehmen (*drag*), über den Bildschirm bewegen und in der gewünschten Anwendung fallenlassen (*drop*).

## 3.2 Die StarView Klassenhierarchie

Abbildung 2.2 zeigt die Klassenhierarchie von StarView. Sie umfaßt z.Zt. dieser Arbeit mehr als 100 Klassen mit mehr als 500 Funktionen.

### 3.2.1 Klassenübersicht nach Kategorie

In der Tabelle 3.1 sind die am häufigsten benutzten StarView Klassen, eingeteilt nach Funktionsgruppen, aufgeführt.

| Kategorie | Klassen |
|---|---|
| Applikation | *application* |
| Fenster | *window, systemWindow, workWindow* |
| Ausgabe | *outputDevice* |
| Ausgabewerkzeuge | *pen, brush, font, gdiMetaFile* |
| Eingabegeräte | *mouseEvent, keyEvent, keyCode* |
| Menü | *menu* (*menuBar, popupMenu*) |
| Accelerator | *accelerator* |
| Interaktionselemente (*controls*) und Dialogboxen | *control, button, edit, listBox, comboBox, scrollBar, fixedText, fixedBitmap, fixedIcon, groupBox, dialog, modalDialog* |
| Druck- und Druckvorschau | *printer, preview* |
| Zwischenablage | *clipboard* |
| Hilfe | *help, helpEvent* |
| Resource | *resource* |

Tabelle 3.1: Klassenübersicht nach Kategorie

### Applikation

Mit der Funktion *execute* aus der Klasse *application* wird die Anwendung gestartet.

### Fenster

Die wichtigsten Klassen zur Erstellung von Fenstern sind *window*, *systemWindow* und *workWindow*. Diese Klassen besitzen neben einer Ausgabefunktionalität Funktionen für die Benutzerinteraktion mit dem Fenstersystem und Funktionen zur Verarbeitung von Eingaben. Die Klasse *window* stellt die ereignisbearbeitenden Funktionen (*event handler*[1]) für die Eingaben in Form virtueller Methoden zur Verfügung. Diese *event handler* können je nach Bedarf überschrieben werden.

### Ausgabe

Die Klasse *outputDevice* stellt die Ausgabefunktionen als Basis für eine geräteunabhängige Ausgabe zur Verfügung. Sie ist Basisklasse für alle Klassen, die Ausgaben darstellen können, z.B. für *window, printer*.

[1] Unter dem Begriff *handler* ist ein Wert zu verstehen, durch den auf ein Objekt zugegriffen werden kann. Frei übersetzt bedeutet *handler* so etwas wie "Griff (Türgriff)" oder "etwas handhaben" (von *to handle*).

### Ausgabewerkzeuge

Alle Ausgabefunktionen der Klasse *outputDevice* arbeiten mit dem Ausgabewerkzeugen *pen, brush* und *font*. Die Klasse *pen* repräsentiert einen Stift für Ausgabefunktionen. *Pens* können mit beliebiger Farbe und Strichstärke erzeugt werden. Die Klasse *brush* repräsentiert ein Muster für Ausgabefunktionen. Es gibt *brushes* mit einem vordefinierten Füllmuster oder mit einem benutzerspezifizierten Muster, das in Form einer *Bitmap* angegeben wird. Die Klasse *font* wird zur Ausgabe von Text verwendet. Mit der Klasse *gdiMetaFile* ist es möglich, alle Ausgaben in einem Ausgabemedium aufzuzeichnen und wieder abzuspielen.

### Eingabegeräte

Die Klassen *mouseEvent, keyEvent* und *keyCode* stellen Funktionen zum Erfragen von Informationen über erfolgte Eingaben durch die Maus oder die Tastatur zur Verfügung.

### Menüs

Die Klasse *menu* bietet Funktionen für die Verwaltung von Menüeinträgen sowie die Zuordnung von Menüeinträgen zu *Popup-Menus* als auch Funktionen für die Verwaltung von *Menu Handlern*. Durch *Popup-Menus* kann eine kontextbezogene Benutzerführung realisiert werden.

### Acceleratoren

Funktionen einer Anwendung können nicht nur über Menüs ausgewählt werden, sondern auch über zugeordnete Tasten oder Tastenkombinationen (*accelerators*). Drückt der Benutzer die entsprechende Taste, wird der *Accelerator Handler* aufgerufen. Die Taste wird anschließend nicht als Tastatureingabe an die Anwendung weitergeleitet.

### Interaktionselemente und Dialogboxen

*Controls* sind Elemente zur Ein- und Ausgabe, die hauptsächlich in Dialogboxen verwendet werden.
Die Gruppe der Control-Klassen ist sehr umfangreich (siehe Abbildung 2.2). In aktiven Controls können Benutzer Eingaben vornehmen. Zu den aktiven Controls gehören die Klasse *button* sowie die davon abgeleiteten speziellen Knopf-Klassen, die Klasse *edit* sowie deren abgeleitete Klassen, die Klassen *listbox* und *multiListBox*, die Klasse *comboBox* mit den davon abgeleiteten Klassen, die Klassen *scrollBar* und *autoScrollBar* sowie benutzerdefinierte Controls. Zu den statischen Controls, die nur zur Anzeige dienen und keine Eingaben entgegennehmen, gehören die Klassen *fixedText, fixedBitmap, fixedIcon* und *groupBox*.
Die Klasse *dialog* stellt die Funktionalität für eine spezielle Fensterform zur Verfügung. In einem Dialog (Dialogbox) sind mehrere Interaktionselemente zusammengefaßt. Generell wird zwischen modalen und nichtmodalen Dialogboxen unterschieden. StarView bietet eine Anzahl von vordefinierten modalen Dialogen: die Messageboxen.

### Druck und Druckvorschau

Die Klassen *printer* und *preview* stellen die notwendige Funktionalität für die Ausgabe auf einen Drucker bzw. für eine möglichst genaue Druckvorschau (*preview*) auf dem Bildschirm zur Verfügung. Diese Klassen enthalten Funktionen zur Seiten- und Druckjobsteuerung.

### Zwischenablage

Die Klasse *clipboard* stellt eine Art Zwischenspeicher für den Datenaustausch zwischen verschiedenen Anwendungen oder innerhalb einer Anwendung zur Verfügung. Im Zwischenspeicher kann nur ein Datenobjekt zur Zeit stehen.

### Hilfe

Zur Steuerung des Hilfesystems gibt es die Klassen *help* und *helpEvent*. Innerhalb einer Anwendung kann damit Hilfe zu einem Schlüsselwort angefordert werden.

### Resourcen

Alle bisher erwähnten Klassen sind von der Klasse *resource* abgeleitet.
Resourcen erlauben es, Informationen über Menüs, Dialogboxen, Zeichenketten usw. getrennt von der Anwendung zu verwalten. Dadurch ist es möglich, existierende Programme zu modifizieren, ohne sie erneut übersetzen zu müssen. Es werden nur die betreffenden Resource-Dateien übersetzt und an die Anwendung gebunden.

## 3.3 Grundelemente einer StarView-Anwendung

Dieser Abschnitt soll anhand eines einfachen Beispiels (siehe Abbildung 3.1) erläutern, welche Elemente mindestens benötigt werden, um eine graphische Oberfläche mit StarView zu entwickeln[2].

### Erzeugen des Hauptfensters

Generell kann eine Anwendung mehrere Arten von Fenstern aufweisen. Ein Fenster ist das Hauptfenster. Wird dieses Fenster geschlossen, so terminiert die gesamte Anwendung. Das Hauptfenster wird mit einem Konstruktor des Abstrakten Datentyp *workWindow.T* und einem Attribut (*stylebit*) WB_APP (das das Fenster zum Hauptfenster erklärt) erzeugt. Die Attribute für Fenster sind in der Klasse *window* zu finden.

Nach dem Erzeugen, der Initialisierung und dem Sichtbarschalten des Fensters mit *window.show(...)* wird die Ereignisverarbeitung (*event-loop*) (s. Abschnitt 3.3) gestartet.

[2]C++ Kode zu diesem Beispiel befindet sich im Anhang A, der vollständige TL Kode in Anhang B.1.

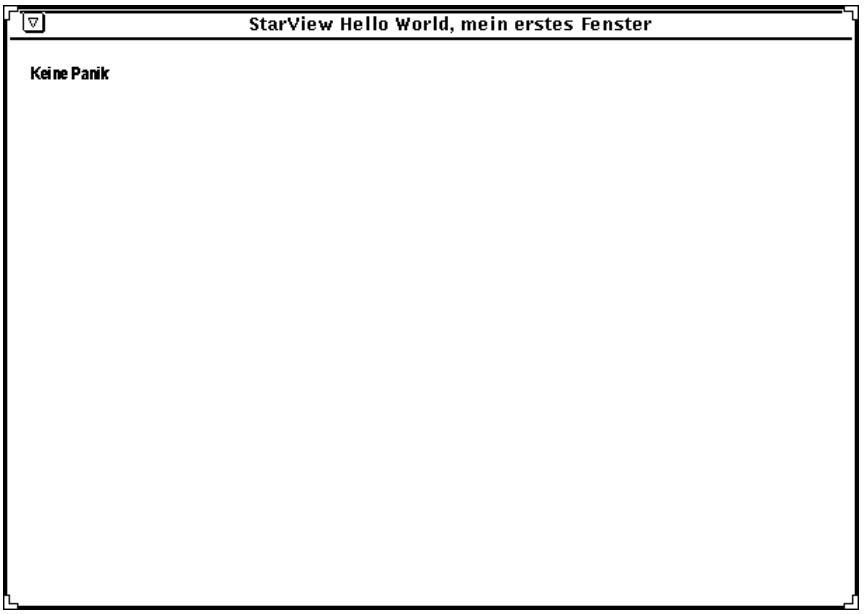


Abbildung 3.1: Ein erstes StarView Beispiel

```
begin
 (* create window *)
  let aMainWin = workWindow.new2(window.nilT
                                 int.bitOr(window.WB_STDWORK window.WB_APP))
 ...
 (* set window header text *)
 window.setText(aMainWin "StarView Paint")
 (* display window *)
 window.show(aMainWin)
 (* start event loop *)
 application.execute()
end
```

Neben dem Grundelement "Hauptfenster" einer StarView Anwendung enthält dieses Programm eine überschriebene Funktion (siehe folgenden Abschnitt über Ereignisbearbeitung) zur Ausgabe eines Textes im Fenster. Zur Ausgabe des Textes wird die Funktion *drawText* aus der Klasse *outputDevice* unter Angabe der Koordinaten und des Textes benutzt.

```
(* define function paint and install it in win *)
window.setPaint(aMainWin
    fun(pEvt :rectangle.T) :Ok
      begin
```

```
    outputDevice.drawText(aMainWin  point.new3(50 50)  "Keine Panik")
   end)
```

### Ereignisbearbeitung

Jedes Ereignis, z.B. eine Benutzereingabe oder das Auftreten eines Fehlers, veranlaßt StarView, eine benutzerdefinierte Funktion (*event handler*) aufzurufen. Der Benutzer erhält die Möglichkeit, auf die Ereignisse zu reagieren und so den weiteren Verlauf der Anwendung zu bestimmen.
Die Anbindung eines *event handler* an ein Ereignis kann durch Überschreiben (*overriding*) der virtuellen Funktion einer abgeleiteten Klasse oder über einen von StarView gestellten Mechanismus (über die Klasse *link*) erfolgen. Das Überschreiben von virtuellen Funktionen ist ein C++-Konzept. Die Funktionen werden per Klasse überschrieben, in TL erfolgt dies per Objekt. Um das Überschreiben von Funktionen in TL zu simulieren, wird jede virtuelle Funktion als Attribut eines Abstrakten Datentyps betrachtet, für das ein Paar aus Schreib- und Lesefunktion existiert. Beispiel: *window.setPaint, window.getPaint.*

Die Möglichkeit des Überschreibens wird hauptsächlich für virtuelle Funktionen der Klasse *window* benutzt, von der der Programmierer häufig Klassen ableitet. Im folgenden Beispiel wird der *event handler MouseButtonDown* überschrieben, um die Position des Mauszeigers im Fenster ermitteln zu können.

```
window.setMouseButtonDown(aMainWin (* Object *)
    fun(mEvt :mouseEvent.T) :Ok
      begin
       window.captureMouse(aMainWin)
       t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
      end)
```

Ist ein *event handler* in einer abgeleiteten Klasse nicht überschrieben worden, so ruft StarView die Standardimplementierung der Funktion der Basisklasse auf.

Die Anbindung von *event handlern* über die Klasse *link* ist für ereignisbearbeitende Funktionen (*handler*) solcher Klassen, von denen der Programmierer nicht unbedingt eine Klasse ableiten muß (z.B. Knöpfe, Rollbalken). Die folgende Funktion reagiert benutzerdefiniert auf ein Ereignis. Diese Bearbeitung des Ereignisses wird mit einem *link* an das betreffende Objekt gebunden (hier an ein Menü).

```
let colorSelect(win :T  pMenu :menu.T) :link.Handler =
      fun() :Ok
        begin
        ...
        end
```

Alle Klassen sind abgeleitet von der Klasse *linkHdl* (siehe Abbildung 2.2).

### Starten der Anwendung

Der Ablauf nach Starten der Anwendung ist in der Abbildung 3.2 schematisch dargestellt. Die Funktion *main* ruft nach einigen Initialisierungen die Funktion *application.execute* auf. Per *ccall* wird StarView aufgerufen und das System befindet sich nun in der Ereignisverarbeitung (*event-loop*). Erfolgt ein Ereignis (*event*), z. B. Mausklick auf eine Menüleiste, so wird per *callback* die ereigniszugehörige TL-Funktion aufgerufen. Die TL-Funktion, die wiederum StarView-Funktionen aufruft, realisiert das per *ccall*. Die graphischen Elemente werden mittels eines Graphik-Servers angezeigt. Die Funktion *systemWindow.close* (bzw. ein Mausklick auf den *quit*-Knopf im Fenstermenü) beendet die *event-loop* und kehrt zur *main*-Funktion zurück.

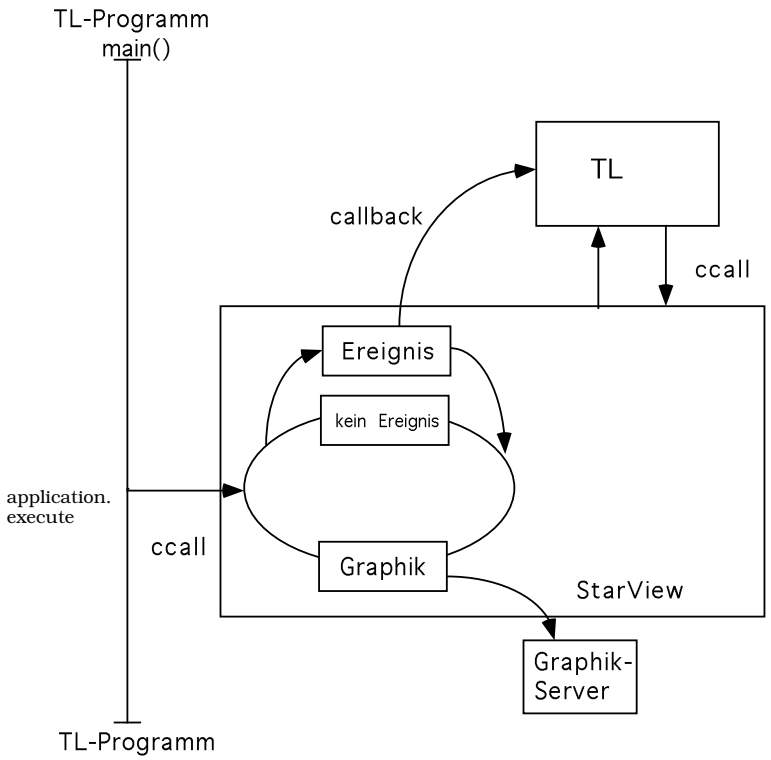


Abbildung 3.2: Die Ereignisverarbeitung (*event-loop*)

# 4. Entwicklung graphischer Benutzeroberflächen mit Tycoon und StarView

Die Erstellung einer graphischen Benutzeroberfläche kann auf drei Wegen erfolgen:

▷ implizit: Generische Editoren zeigen Daten an.
▷ explizit: Die Funktionen der Graphikbibliotheken werden direkt angesprochen.
▷ interaktiv: Der Aufbau der Oberfläche wird mit einem Werkzeugkasten unterstützt.

Der explizite Weg erfolgt bei Tycoon/StarView über die Bibliothek *svenv*. Über die Bibliothek *sveditenv* ist der implizite Weg gewählt. Eine interaktive Unterstützung liefert der Design Editor von StarView. Es läßt sich pauschal nicht ein Weg als der beste festlegen. Je nach Komplexität der Fragestellung läßt sich entscheiden, welcher Weg für welches graphische Problem geeignet ist.
Dieses Kapitel stellt die drei Wege vor, wobei der Schwerpunkt auf dem expliziten Weg liegt.

## 4.1 Der implizite Weg

Die Bibliothek *sveditenv* ist der Nachfolger der Editorbibliothek *editenv* [Müßig 94]. Sie ermöglicht das schnelle Entwerfen von typsicheren generischen Editoren. Allen so entworfenen Editoren ist ein eine einheitliche Präsentation (*look*) und Dialogsteuerung (*feel*) gemeinsam. Diese Editoren sind beliebig kombinierbar und zeigen sowohl atomare als auch hierarchisch strukturierte Werte des Tycoon-Objektspeichers an. Die Werte in den Editoren können vom Anwender verändert werden. Die Menge der Editoren läßt sich in drei Klassen einteilen: die Basiseditoren, die Verbundeditoren und die Massendateneditoren. Tycoon Basistypen werden mit Basiseditoren, Tycoon Typkonstruktoren mit Verbundeditoren und Massendaten mit Massendateneditoren angezeigt. In Abschnitt 4.1.2 findet sich ein Beispiel für einen Basiseditor für Zeichenketten. Abbildungen anderer Editoren finden sich in [Müßig 94].
Das *editenv* basiert auf NeWS. Das *sveditenv* auf StarView. Die Abbildung 4.1 zeigt, wo diese beiden Bibliotheken einzuordnen sind. Die Schnittstellen der beiden Editorbibliothe-

ken sind sich sehr ähnlich. Kleine Unterschiede in den Funktionen der Bibliothek *sveditenv* zu *editenv* sind in der Verwendung der StarView Bibliothek begründet. Die Darstellung der Editoren nach einem erfolgten Funktionsaufruf unter Verwendung der *sveditenv* ist nicht ganz identisch mit der Darstellung der Editoren, die mit *editenv* erzeugt worden sind, vom Grundaufbau jedoch ähnlich.

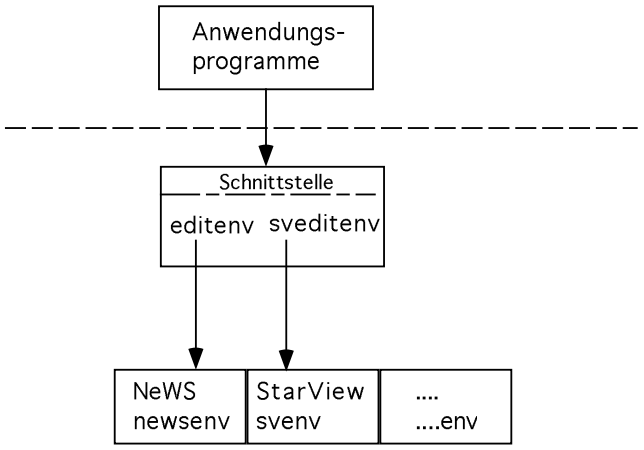


Abbildung 4.1: Der implizite Weg

### 4.1.1 Aufbau der Editoren

Alle Editoren besitzen einen gleichen Rahmen, der eine Basisfunktionalität zur Verfügung stellt. Neben einer Titelzeile befinden sich in jedem Editor eine feste Anzahl von Knöpfen. Die oberen drei Knöpfe lassen sich als Menü auffassen. Welche Einträge, Subeinträge und Funktionalität sich hinter den Knöpfen verbirgt ist in Tabelle 4.1 aufgeführt und erläutert. Nicht alle Funktionen stehen allen Editoren zur Verfügung. Nicht verfügbare Funktionen sollen vom Anwender nicht ausgewählt werden können und sind deshalb als inaktiv markiert.

Die unteren zwei Knöpfe (der *Apply*- und *Reset*-Knopf) dienen dem Datenaustausch zwischen Anwendung und Editor. Mit dem *Apply*-Knopf wird der aktuelle Wert aus dem Editor in die Datenstruktur bzw. Variable unter Einhaltung von eventuellen Integritätsbedingungen zurückgeschrieben. Der *Reset*-Knopf liest den aktuellen Wert aus der betreffenden Datenstruktur bzw. Variablen heraus und zeigt ihn in dem Editor an.

### 4.1.2 Beispiel eines Basiseditors

Dieses Beispiel erzeugt auf einfache Weise einen Editor für Zeichenketten. Das nachfolgende Tycoon-Kodefragment erzeugt den in Abbildung 4.2 dargestellten Editor für Zeichenketten.

| Menü | Eintrag | Subeintrag | Funktionalität |
|---|---|---|---|
| Quit | | | Schließt Editor mit Nachfrage. |
| Edit | Edit | | Öffnet Editor mit ausgewähltem Element. |
| | New | New | Fügt Element an aktueller Position ein. |
| | | New Behind | Fügt Element hinter aktueller Position ein. |
| | Cut | Cut | Löscht aktuelles Element bzw. setzt Grundwert. |
| | | Cut All | Löscht alle Elemente. |
| | | Cut Selected | Löscht ausgewählte Elemente. |
| | Copy | | Kopiert in Puffer. |
| | Paste | | Kopiert aus Puffer. |
| View | Properties | | Grundeinstellungen setzen. |
| | Path... | | Zeigt Pfad zum Editor an. |

Tabelle 4.1: Funktionalität des Menüs der Editoren

```
import
  editor (* from sveditenv *)

let var name = "Meyer"
let var continue = true
let quit = fun() continue := false
let dispatch = fun() continue

let stringEditor = editor.newString("My first Editor"
   let get = fun() name
   let set = fun(new :String) name := new)

editor.display(stringEditor quit editor.somewhere) (* display editor somewhere *)
editor.waitAndDispatch(dispatch)  (* start event loop *)
```

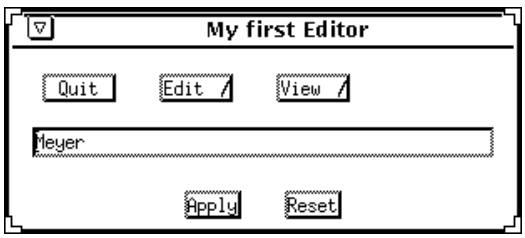


Abbildung 4.2: Editor zum Anzeigen von Zeichenketten

### 4.1.3 Aufbau der Editorbibliothek

Die Editorbibliothek *sveditenv* umfaßt acht Module, die sich vier Schichten zuordnen lassen (siehe Abbildung 4.3). Die Abbildung gibt nicht die komplexen Importbeziehungen innerhalb der Editorbibliothek wieder.

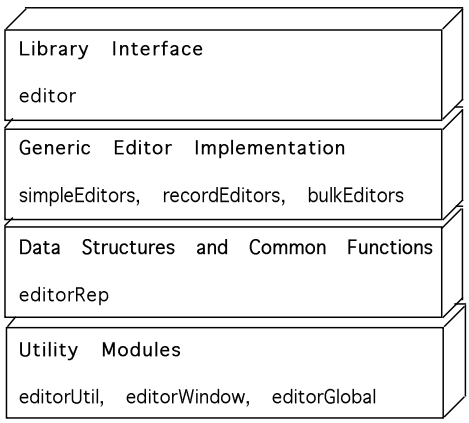


Abbildung 4.3: Struktur der Editorbibliothek

Die oberste Schicht (*Library Interface*) bildet das Modul *editor*; nur über dieses Modul ist ein Zugriff auf die Editorbibliothek möglich. Es erhält alle wichtigen Definitionen der anderen Module. In der *Generic Editor Implementation*-Schicht befinden sich die Module *simpleEditors, recordEditors* und *bulkEditors*, die eigentlichen Implementationen der Editoren. Ein Protokoll, über das die Editoren angesprochen werden, ist in der *Data Structures and Common Functions*-Schicht mit dem Modul *editorRep* definiert. In der untersten Schicht (*Utility Modules*) finden sich die Hilfsmodule *editorUtil, editorWindow* und *editorGlobal*. Eine genaue Beschreibung der Module findet sich in [Müßig 94].

## 4.2 Der explizite Weg

Beim expliziten Weg (siehe Abbildung 4.4) greift das Anwendungsprogramm direkt auf die StarView-Bibliothek zu. Es stehen alle Klassen und Funktionen zur Verfügung.
Die folgenden Beispiele sollen Grundlagen für eine Programmierung in TL unter Verwendung der StarView Bibliothek liefern. Die Beispiele sind dem StarView Handbuch [Busch et al. 93] entnommen und an TL angepaßt worden. Die ersten drei Beispiele bauen aufeinander auf. Der vollständige Quelltext der Beispiele findet sich in Anhang B.

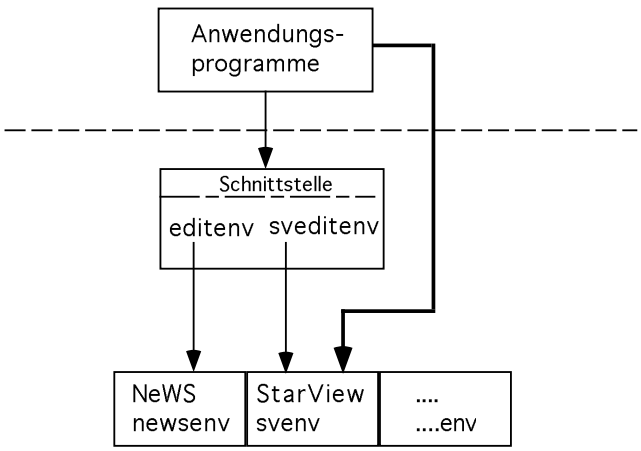


Abbildung 4.4: Der explizite Weg

### 4.2.1 HelloWorld

Dieses Beispiel wurde bereits in Abschnitt 3.3 schon einmal kurz vorgestellt.
Das Hauptfenster wird mit dem Konstruktor *new2* aus der Klasse *workWindow* erzeugt. Das eigentliche Fenster wird in der Funktion *main* angelegt, wobei das *stylebit WB_APP* dafür sorgt, daß das Fenster zum Hauptfenster der Anwendung wird. Durch das *stylebit WB_STDWORK* erhält das Hauptfenster alle für ein Hauptfenster möglichen Eigenschaften: es ist verschiebbar, in der Größe veränderbar, minimierbar, maximierbar und schließbar. Der Fenstertext wird über die Funktion *setText* aus der Klasse *window* gesetzt. Mit der Funktion *show* aus der Klasse *window* wird das Fenster sichtbar geschaltet. Die Ereignisverwaltung (*event loop*) wird mit der Funktion *execute* aus der Klasse *application* gestartet.

```
let aMainWin = workWindow.new2(window.nilT
                               int.bitOr(window.WB_STDWORK window.WB_APP))
window.setText(aMainWin "StarView Hello World, mein erstes Fenster")
(* make window visible *)
window.show(aMainWin)
(* start event loop *)
application.execute()
```

Damit innerhalb des Fensters der Text "Keine Panik" erscheint, muß die Funktion *paint* aus der Klasse *window* überschrieben werden. Das Überschreiben wird in TL durch die Funktion *setPaint* simuliert.
Innerhalb der überschriebenen Funktion wird die Ausgabefunktion *outputDevice.drawText* mit den Koordinaten und dem gewünschten Text aufgerufen. Da noch kein Koordinatenpunkt existiert, wird einer mit *new3* aus der Klasse *point* konstruiert.

```
window.setPaint(aMainWin
    fun(pEvt :rectangle.T) :Ok
      begin
        outputDevice.drawText(aMainWin point.new3(50 50) "Keine Panik")
      end)
```

### 4.2.2 Painter

Dieses Beispiel (siehe Abbildung 4.5) zeigt neben den Grundelementen einer StarView-Anwendung ein Menü, Verarbeitung von Mauseingaben und die Ausgabe mit einem der Ausgabewerkzeuge der Klasse *outputDevice* - dem *Pen*.

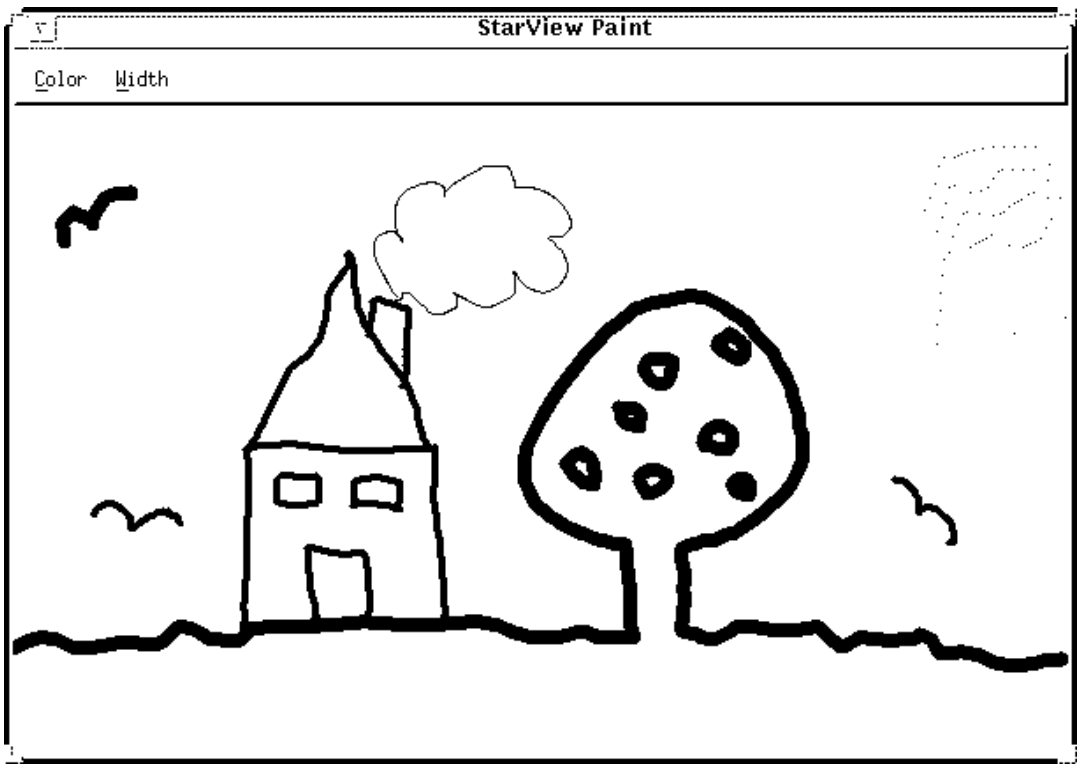


Abbildung 4.5: Das Hauptfenster des Beispiels "Painter"

Dieses Programm bietet folgende Funktionalität:

- Der Benutzer kann mit Hilfe der Maus Linien in dem Fenster zeichnen.
- Für den *Pen* kann über ein Menü eine Farbe und eine Strichbreite ausgewählt werden.

- Nach der Fenstermenüwahl *quit* ist über eine Messagebox zu entscheiden, ob die Anwendung tatsächlich beendet werden soll.

Inhaltlich ergeben sich vier Teilbereiche, die im folgenden erläutert werden:

- Aufbau und Umgang mit Menüs
- Verarbeitung von Mauseingaben
- Realisierung des Zeichenvorgangs
- Realisierung der Messagebox

**Das Menü:**

Menüs sollen den Benutzer in der Kommunikation mit der Anwendung unterstützen. Ein Menü enthält Einträge, die die Möglichkeit der Einflußnahme auf den Ablauf der Anwendung aufzeigen. Nach Auswahl eines Eintrages wird die damit verbundende Aktion ausgelöst.
Zunächst sichtbar ist die Menüleiste, die nach dem Erzeugen und Zuordnen zu einer Anwendung am oberen Rand unter der Titelzeile des Hauptfensters (beim Mac am oberen Rand des Bildschirms) erscheint. Die Menüleiste (siehe Abbildung 4.6) enthält im Beispiel die Menü-Einträge (*Items*) *Color* und *Width*. Die unterstrichenen Buchstaben "C" und "W" sind Tastenkürzel, mit denen die Einträge mit der Tastatur statt der Maus ausgewählt werden können.

Abbildung 4.6: Die Menüleiste des Beispiels "Painter"

Beide Menüeinträge besitzen Popupmenüs.

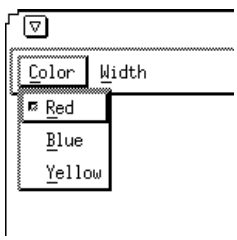


Abbildung 4.7: Das Popupmenü *Color*

Zunächst werden die Popupmenüs und die Menüleiste (siehe Abbildung 4.7) erzeugt:

```
let aColorMenu = menu.newPopup1()
let aWidthMenu = menu.newPopup1()
let aMenuBar = menu.newBar1()
```

Über benutzerdefinierte Konstanten (*Identifiers, ID's*) für Menüeinträge kann auf die Einträge zugegriffen werden:

```
let MID_PENCOLOR    = 10
let MID_PENRED      = 11
let MID_PENBLUE     = 12
let MID_PENYELLOW = 13

let MID_PENWIDTH   = 20
let MID_PENSMALL   = 21
let MID_PENMEDIUM = 22
let MID_PENTHICK   = 23
```

Einträge in Menüleisten erfolgen über die Funktion *insertItem*, Folge-Menüeinträge in Popupmenüs über die Funktion *insertItem2*. Die Konstante *MENU_APPEND* sorgt dafür, daß neue Folge-Menüeinträge (in Popupmenüs) an den bisher letzten Folge-Menüeintrag gehängt werden. Menüeinträge sind markierbar, wenn das Attribut *MIB_CHECKABLE* angegeben ist. Die Tastenkürzel werden mit einer Tilde vor dem betreffenden Buchstaben festgelegt. Alle genannten Funktionen und das Attribut stammen aus der Klasse *menu*.

```
menu.insertItem(aMenuBar MID_PENCOLOR "~Color")

...

menu.insertItem2(aColorMenu MID_PENRED "~Red"
                 menu.MENU_APPEND menu.MIB_CHECKABLE)
menu.insertItem2(aColorMenu MID_PENBLUE "~Blue"
                 menu.MENU_APPEND menu.MIB_CHECKABLE)
menu.insertItem2(aColorMenu MID_PENYELLOW "~Yellow"
                 menu.MENU_APPEND menu.MIB_CHECKABLE)
```

Popupmenüs werden mit der Funktion *changePopupMenu* aus der Klasse *menu* der Menüleiste zugeordnet. Die Funktion *changeAppMenu* aus der Klasse *application* erklärt ein Menü zur Menüleiste der Anwendung.

```
menu.changePopupMenu(aMenuBar MID_PENCOLOR aColorMenu)
...
application.changeAppMenu(aMenuBar)
```

Die Funktion *pushSelectHdl* aus der Klasse *menu* bindet über einen *link* die benutzerdefinierten Funktionen *colorSelect* und *widthSelect* an den Aufruf aus der Menüleiste.

```
let colorSelect(win :T  pMenu :menu.T) :link.Handler =
  fun() :Ok
   ...    (* realization of menu item Color *)

menu.pushSelectHdl(aColorMenu
                   link.new(colorSelect(aMainWin aColorMenu)))
```

Sind Menüeinträge als markierbar definiert (über das Attribut *MIB_CHECKABLE*), besteht die Möglichkeit, mit der Funktion *checkItem* aus der Klasse *menu* und Angabe der ID, diesen Menüeintrag mit einer Markierung zu versehen. Mit dem gesetzten Parameter *false* in der Funktion *checkItem2* aus der Klasse *menu* werden bestehende Markierungen gelöscht. Die Funktion *getCurItemId* aus der Klasse *menu* liefert die benötigte ID des momentan selektierten Menüeintrags.

```
let colorSelect(win :T  pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
      (* if there is a mark, delete it *)
      if win.nColor != 0 then
         menu.checkItem2(pMenu win.nColor false)
      end
       (* the tuple component nColor gets the ID of chosen color *)
       win.nColor:= menu.getCurItemId(pMenu)
       (* the chosen color is checked *)
       menu.checkItem(pMenu win.nColor)
      ...
    end
```

**Die Mauseingaben:**

Neben der Tastatur ist die Maus gerade bei Anwendungen mit Fenstern ein wichtiges Eingabemedium. Zu den Aktionen einer Maus gehören der Mausklick mit einer Maustaste und das Bewegen und Positionieren des Mauszeigers in einer Anwendung, z.B. auf einen Knopf. Nicht alle Plattformen unterstützen alle drei Maustasten (der Mac z.B. unterstützt nur eine Maustaste). StarView bietet hier Funktionen, die die Besonderheiten der Systeme sowie der verwendeten Maustypen berücksichtigt. Alle Aktionen können unter allen Voraussetzungen ausgeführt werden.
In dem Beispielprogramm hat der Benutzer die Möglichkeit, mit Hilfe der Maus Linien in dem Anwendungsfenster zu zeichnen. Bevor der Zeichenvorgang ausgelöst wird, müssen erst die vom Benutzer erfolgten Mausaktionen von der Anwendung verarbeitet werden.

Die Funktionen, die die Mausaktionen festlegen, müssen in der Klasse *window* überschrieben werden (siehe Abschnitt 3.3 über Ereignisbearbeitung). Von Interesse in diesem Beispiel

sind der *MouseButtonDown-Handler* und der *MouseMove-Handler*. Der *MouseButtonDown-Handler* wird gerufen, wenn eine Maustaste gedrückt wird, der *MouseMove-Handler* bei jeder Mausbewegung über dem Fenster.
In der vom Programmierer überschriebenen Funktion *MouseButtonDown* muß die Position des Mauszeigers im Fenster ermittelt werden, damit Anfangs- und Endpunkt der zu zeichnenden Linie bestimmt werden kann:

```
window.setMouseButtonDown(aMainWin
  fun(mEvt :mouseEvent.T) :Ok
    begin
      ...
      t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
    end)
```

Mit der überschriebenen Funktion *MouseMove* wird überprüft, ob es sich bei der Mausbewegung um die gewünschte Art der Bewegung – Bewegung mit gedrückter Taste – handelt:

```
window.setMouseMove(aMainWin
  fun(mEvt :mouseEvent.T) :Ok
    if mouseEvent.isLeft(mEvt) then
    ...
    end)
```

### Die Ausgabe (Realisierung des Zeichenvorgangs):

Alle Ausgabefunktionen stellt die Klasse *outputDevice* zur Verfügung. Diese Ausgabefunktionen arbeiten mit verschiedenen Ausgabewerkzeugen, für die eigene StarView-Klassen existieren. Für Textausgaben wird die Klasse *Font* verwendet, für Graphik (Punkte, Linien, Figuren etc.) die Klasse *Pen* und zum Ausmalen von Flächen mit einer Farbe und einem Muster die Klasse *Brush*.
In diesem Beispiel wird zum Zeichnen einer Linie zwischen zwei Punkten die Funktion *drawLine* aus der Klasse *outputDevice* verwendet. Die Ausgabe erfolgt über das Ausgagewerkzeug *Pen*.
Ein *Pen* ist ein Zeichenstift, der Spuren in einem Fenster hinterläßt. *Pens* haben drei Attribute: Strichbreite, Farbe und Linientyp (durchgezogen, gestrichelt etc.).

Der *Pen* erhält durch die Funktionen *pen.changeColor* und *pen.changeWidth* die über das Menü ausgewählte Farbe und Breite. Durch die Funktion *outputDevice.changePen* wird der neu ausgewählte *Pen* der aktuelle *Pen*. Als Farbe wird ein Objekt der Klasse *color* übergeben:

```
let colorSelect(win :T  pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
      if win.nColor != 0 then
```

```
    menu.checkItem2(pMenu win.nColor false)
  end
   win.nColor:= menu.getCurItemId(pMenu)
   menu.checkItem(pMenu win.nColor)
  let aPen = outputDevice.getPen(win.win)
  (* if chosen color red then *)
  if win.nColor == MID_PENRED then
    (* change pencolor to red *)
    pen.changeColor(aPen color.new2(color.COL_RED))
  elsif win.nColor == MID_PENBLUE then
    pen.changeColor(aPen color.new2(color.COL_BLUE))
  else
    pen.changeColor(aPen color.new2(color.COL_YELLOW))
  end
   (* new chosen color is applied to pen *)
   outputDevice.changePen(win.win aPen)
end
```

**Die Messagebox:**

Häufig ergeben sich Situationen in einer Anwendung, die eine Nachricht an den Benutzer erforderlich machen, z.B. wenn nicht genügend Speicherplatz vorhanden ist, eine angegebene Datei nicht existiert, bei Sicherheitsabfragen etc. Damit sichergestellt ist, daß der Benutzer die Nachricht auch erhält, wird die Anwendung unterbrochen, eine Messagebox angezeigt und erst nach Beantworten der Messagebox wieder freigegeben. Messageboxen sind daher modale Dialogboxen. StarView stellt verschiedene Messageboxen zur Verfügung. Die *InfoBox* für Informationszwecke, die *WarningBox* für Warnungen an den Benutzer, die *ErrorBox* für Fehlermeldungen und die *QueryBox* für Nachfragen. Die Boxen unterscheiden sich im Informationssymbol und Anzahl der Knöpfe. Die Messagebox in Abbildung 4.8 zeigt das Informationssymbol "?" und zwei Knöpfe
Dieses Beispiel benutzt eine Messagebox, um nachzufragen, ob die Anwendung tatsächlich beendet werden soll. Es wird also ein Objekt der Klasse *queryBox* erzeugt. Diese *QueryBox* enthält einen *Ok*- und einen *Cancel*-Knopf (vergleiche Abbildung 4.8).

Die Messagebox wird mit der Funktion *messBox.execute* aktiviert. Diese Funktion liefert an die Anwendung den Kode des Knopfes, der vom Benutzer ausgewählt wurde. In diesem Beispiel wurde der OK-Knopf mit der Funktion *systemWindow.setClose* verbunden, wodurch die Anwendung mit einer Nachfrage beendet wird. Diese Funktion überschreibt die Funktion *systemWindow.close*, die trotzdem innerhalb der neuen definierten Funktion mit alter Funktionsweise aufgerufen werden kann.

```
systemWindow.setClose(t.win
    fun() :Bool
```

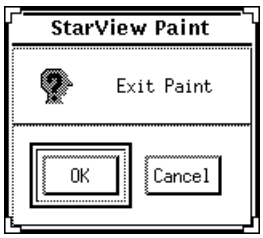


Abbildung 4.8: Die Messagebox zum Verlassen von "Painter"

```
begin
  let aBox =
    queryBox.new(t.win  int.bitOr(window.WB_OK_CANCEL window.WB_DEF_OK)
                 "Exit Paint")
  if messBox.execute(aBox) == messBox.RET_OK then
    systemWindow.close(t.win)
  else
    false
  end
end)
```

### 4.2.3 PainterWithAboutBox

Dieses Beispiel (siehe Abbildung 4.9) ist eine Erweiterung des vorigen Beispiels. Zwei weitere StarView-Klassengruppen werden eingeführt: die Dialog-Klassen und die Control-Klassen.

Die Funktionalität vom Beispiel *Painter* bleibt erhalten. Dieses Beispiel wird um einen Eintrag in der Menüleiste (siehe Abbildung 4.10), der zwei Einträge enthält (siehe Abbildung 4.11), erweitert. Wird der Eintrag "About..." ausgewählt, so öffnet sich im Hauptfenster eine *AboutBox* mit verschiedenen farbigen geometrischen Figuren, einem Text und einem Ok-Knopf. Eine aktivierte *AboutBox* wird mit Anklicken des Ok-Knopfs wieder geschlossen. Bei Auswahl des Eintrags "Exit" wird die Anwendung beendet.
Die *AboutBox* (siehe Abbildung 4.12) gehört zu den modalen Dialogboxen. In der *AboutBox* des Beispiels sind zwei Interaktionselemente (*controls*) enthalten: Ein fester Text (hier ein Copyright-Vermerk) und ein Ok-Knopf, der bei Anklicken mit der Maus das Terminieren der Box veranlaßt.

Der feste Text wird mit einer Funktion aus der Klasse *fixedText* erzeugt. Der Ok-Knopf mit einer Funktion aus der Klasse *okButton*. Die Klassen *fixedText* und *okButton* gehören zu den Control-Klassen. *Controls* sind eigenständige Bedienelemente, die einem Eltern-Fenster (*parent window*) (hier der *AboutBox*) zugeordnet sind. Das Eltern-Fenster (die *AboutBox*) ist eine modale Dialogbox und wird mit einem Konstruktor aus der Klasse *modalDialog*

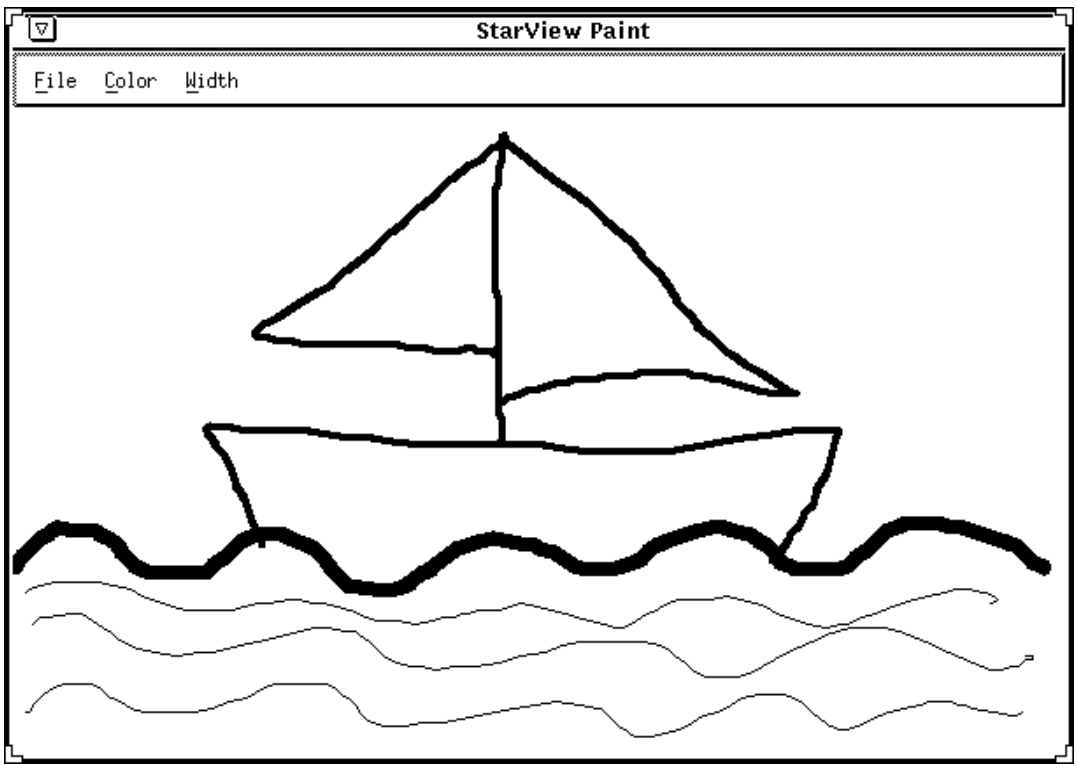


Abbildung 4.9: Das Hauptfenster des Beispiels "PainterWithAboutBox"

erzeugt.

```
let modDiag = modalDialog.new2(parent window.WB_STDMODAL)
let aText = fixedText.new1(modDiag)
let aBut = okButton.new2(modDiag int.bitOr(window.WB_DEFBUTTON
                                           window.WB_TABSTOP))
```

Ein Ok-Knopf ist ein spezielles Interaktionselement, er enthält als Text automatisch "OK". Beim Anklicken des Ok-Knopfs ruft StarView intern die Standardfunktion des Eltern auf. In diesem Beispiel ist der Elter eine modale Dialogbox, somit wird eine Funktion zum Beenden der *AboutBox* aufgerufen. Durch Setzen eines *links* für den Ok-Knopf kann eine andere

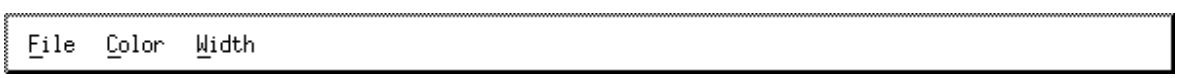


Abbildung 4.10: Die Menüleiste des Beispiels "PainterWithAboutBox"

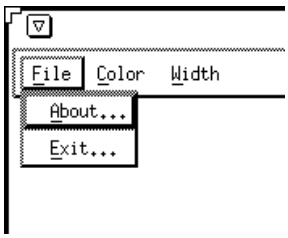


Abbildung 4.11: Das Popupmenü *File* des Beispiels "PainterWithAboutBox"

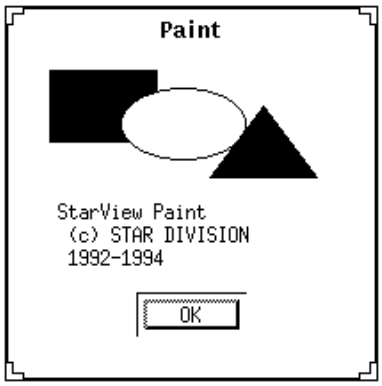


Abbildung 4.12: Die *AboutBox* des Beispiels "PainterWithAboutBox"

Aktion an den Knopf gebunden werden. Damit der Ok-Knopf auch durch Betätigen der Eingabetaste ausgelöst werden kann, wird das Attribut *WB_DEFBUTTON* zugewiesen. Position und Größe sowohl der *AboutBox* als auch des festen Textes und des Ok-Knopfs können, da sie Fenster sind, mit den Funktionen *changePosPixel* und *changeSizePixel* aus der Klasse *window* gesetzt werden. Der Ausgabebereich des Fensters (der *AboutBox*) wird mit der Funktion *window.changeOutputSizePixel* festgelegt. Die Positionsangabe bei der *AboutBox* bezieht sich auf den Bildschirm. Bei den beiden Interaktionselementen (Ok-Knopf und fester Text), die Kindfenster sind, wird die Position relativ zu ihrem Elter (hier der *AboutBox*) interpretiert. Der feste Text erhält über die Funktion *window.setText* seinen Text. Der Ok-Knopf erhält automatisch seinen Text "OK". Die Objekte "fester Text" und Ok-Knopf werden mit der Funktion *window.show* sichtbar geschaltet.

```
window.setText(modDiag "Paint")
window.changeOutputSizePixel(modDiag size.new3(200 180))
window.setText(aText "StarView Paint\n (c) STAR DIVISION\n 1992-1994")

window.changePosPixel(aText point.new3(25 80))
```

```
window.changeSizePixel(aText size.new3(150 50))
window.show(aText)

window.changePosPixel(aBut point.new3(65 135))
window.changeSizePixel(aBut size.new3(70 25))
window.show(aBut)
```

Für die Ausgabe der farbigen Figuren der *AboutBox* wird in diesem Beispiel die Funktion *paint* der Klasse *window* überschrieben. In der neuen *paint*-Funktion werden mit den Funktionen *drawRect, drawEllipse* und *drawPolygon* die drei geometrischen Figuren definiert. Für die Hintergrundfarbe für jede der Figuren wird jeweils ein Objekt der Klasse *brush* erzeugt, das durch die Funktion *outputDevice.changeFillInBrush* zur entsprechend aktuellen Hintergrundfarbe wird.

```
let paint(rect :rectangle.T) :Ok =
   begin
     let aOldBrush =
        outputDevice.changeFillInBrush(modDiag brush.new4(color.new2(color.COL_RED)))
     outputDevice.drawRect(modDiag rectangle.new3(point.new3(20 10)
                                                  point.new3(80 50)))

     outputDevice.changeFillInBrush(modDiag
                                    brush.new4(color.new2(color.COL_YELLOW)))
     outputDevice.drawEllipse(modDiag rectangle.new3(point.new3(60 20)
                                                     point.new3(130 60)))


     outputDevice.changeFillInBrush(modDiag
                                    brush.new4(color.new2(color.COL_GREEN)))
     let poly = polygon.new2(3)
     polygon.changePoint(poly point.new3(110 70) 0)
     polygon.changePoint(poly point.new3(140 30) 1)
     polygon.changePoint(poly point.new3(170 70) 2)
     outputDevice.drawPolygon(modDiag poly)

     outputDevice.changeFillInBrush(modDiag aOldBrush)
   end

window.setPaint(modDiag paint)
```

Das Polygon wird mit Angabe der Anzahl seiner Ecken erzeugt. Mit der Funktion *changePoint* aus der Klasse *polygon* werden die Koordinaten festgelegt.
Das Zuordnen des neuen Popupmenüs (*File*), das Einfügen seiner beiden Einträge ("A-

bout..." und "Exit...") erfolgen analog zu dem im vorigen Beispiel erklärten Verfahren. Bei der Auswahl des Eintrags "About..." wird über den *link* die Funktion *fileSelect* aufgerufen und somit ein Objekt der Klasse *aboutBox* erzeugt und durch die Funktion *modalDialog.execute* gestartet. Bei Auswahl des Eintrags "Exit..." wird die (bereits aus vorigem Beispiel bekannte) Messagebox aktiviert.

```
(* link the function fileSelect *)
menu.pushSelectHdl(aFileMenu
                   link.new(fileSelect(aMainWin aFileMenu)))


let fileSelect(win :T pMenu :menu.T) :link.Handler =
   fun() :Ok
     if menu.getCurItemId(pMenu) == ABOUT then
        modalDialog.execute(about3.newAboutBox(win.win))
        ok
     else
        let aBox = queryBox.new(win.win int.bitOr(window.WB_OK_CANCEL
                                                  window.WB_DEF_OK)
                                "Exit Paint")
       if messBox.execute(aBox) == messBox.RET_OK then
          systemWindow.close(win.win)
       else
          false
       end
     end
```

## 4.3 Der interaktive Weg

Der Design Editor von StarView ist ein Resource Editor (siehe Abschnitt 2.2), der die Möglichkeit bietet, die Resource Dateien einer StarView-Anwendung interaktiv zu erstellen. Eine komplette Anwendung kann nicht mit dem Design Editor erstellt werden. Es ist eine Kombination mit dem expliziten Weg nötig, weil keine graphische Umgebung für den Entwurf von Aktionen auf Ereignisse existiert

Mit dem Design Editor (siehe Abbildung 4.13) sollen besonders einfach Dialogboxen und Menüs erstellt werden können [Busch et al. 93]. Die ausgewählten Resource Objekte werden entweder über Editoren oder Dialogboxen bearbeitet. Die Eigenschaften von Basistypen, wie z.B. String, Pen, Brush oder Font, werden in Dialogboxen festgelegt. Für den Entwurf von komplexeren Resource Objekten, wie z.B. Dialogboxen, Fenster und Menüs, gibt es spezielle Editoren (Abbildung 2.1 zeigt einen Editor zum Bearbeiten von Fenstern), in denen die Objekte und deren Sub-Resourcen (z.B. Ok-Knöpfe, Bildlaufleisten und Textfelder in einem Fenster) bearbeitet werden können.

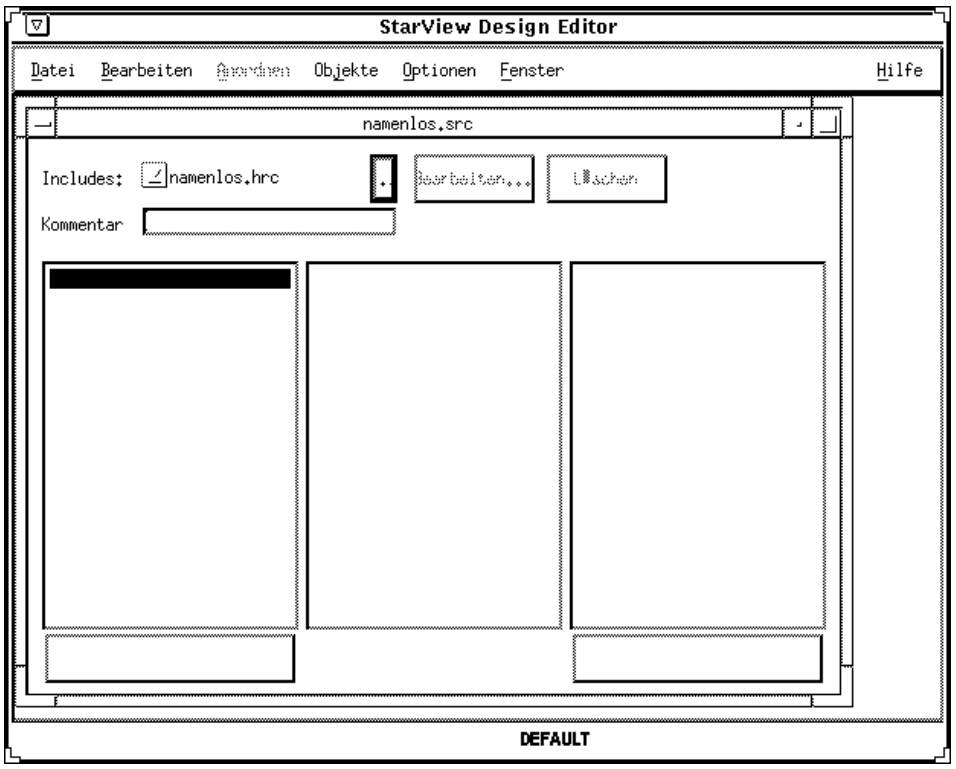


Abbildung 4.13: Der Design Editor

Die Möglichkeit Resourcen in einer Anwendung zu benutzen wird von StarView nicht auf allen Plattformen unterstützt, auf denen auch das Tycoon-System portiert wurde. So ist es nicht möglich, Teile von graphischen Oberflächen, von Tycoon-Anwendungen die auf PC und Mac laufen, aus Resourcen zu laden. Solaris 2.x wird in Bezug auf Resourcen von StarView unterstützt. Damit jedoch Tycoon-Anwendungen mit einer graphischen Oberfläche ohne Probleme auf allen drei Plattformen funktionieren, sind die nicht portablen Funktionen für den Tycoon-Programmierer nicht verfügbar. Die Funktionen werden wieder zugänglich sein, sobald StarView diese Funktionen für alle verwendeten Plattformen zur Verfügung stellt. Das folgende Beispiel führt trotzdem, zur vollständigen Übersicht, in die Arbeit mit Resourcen ein.

### "PainterWithAboutBox" mit Verwendung von Resourcen

Dieses Beispiel führt in die Arbeit mit Resourcen ein. Von der Funktionalität her ist es mit Beispiel "PainterWithAboutBox" (vgl. Absatz 4.2.3) identisch, es erfolgt lediglich eine Umstellung auf Resourcen.
Resourcen erlauben es, bestimmte Daten einer Anwendung getrennt von dieser zu verwal-

ten und zu bearbeiten. Bei Änderung dieser Daten müssen nur die betreffenden Resource-Dateien geändert und neu übersetzt werden. Das schon übersetzte Hauptprogramm muß nicht noch einmal übersetzt werden. Die Resourcen werden mit dem Resource-Compiler übersetzt. Die übersetzten Resourcen werden zur Laufzeit von der Anwendung geladen.
In diesem Beispiel sind die spezifischen Daten des Menüs und der AboutBox mit dem Design Editor entworfen und in je einer Resource-Datei gespeichert worden. Die Resource-Dateien können sowohl mit einem Texteditor als auch mit dem Design Editor erstellt werden.

**Das Menü aus Resourcen heraus geladen:**

Damit aus der Anwendung heraus auf die Resourcen eindeutig zugegriffen werden kann, benötigen sie einen Identifier. Die Resource-Datei für das Menü gibt zuerst die Resource-ID des Menüs (hier *MID_MENUBAR*) an. Für jeden Menüeintrag (*Menu-Item*) folgen der Text des Eintrags und die entsprechende *MenuItem-ID*.

```
Menu MID_MENUBAR
{
   ItemList =
   {
      MenuItem
      {
         Text       = "~File";
         Identifier = MID_FILE;
         ...
      };

      MenuItem
      {
         Text       = "~Color";
         Identifier = MID_PENCOLOR;
         ...
      };

      MenuItem
      {
         Text       = "~Width";
         Identifier = MID_PENWIDTH ;
         ...
      };
   };
};
```

Im Design Editor wird das Menü in einem speziellen Editor bearbeitet (siehe Abbildung 4.14): dem Menü-Editor. Hier können die Menüeinträge vorgenommen werden und bei Bedarf wei-

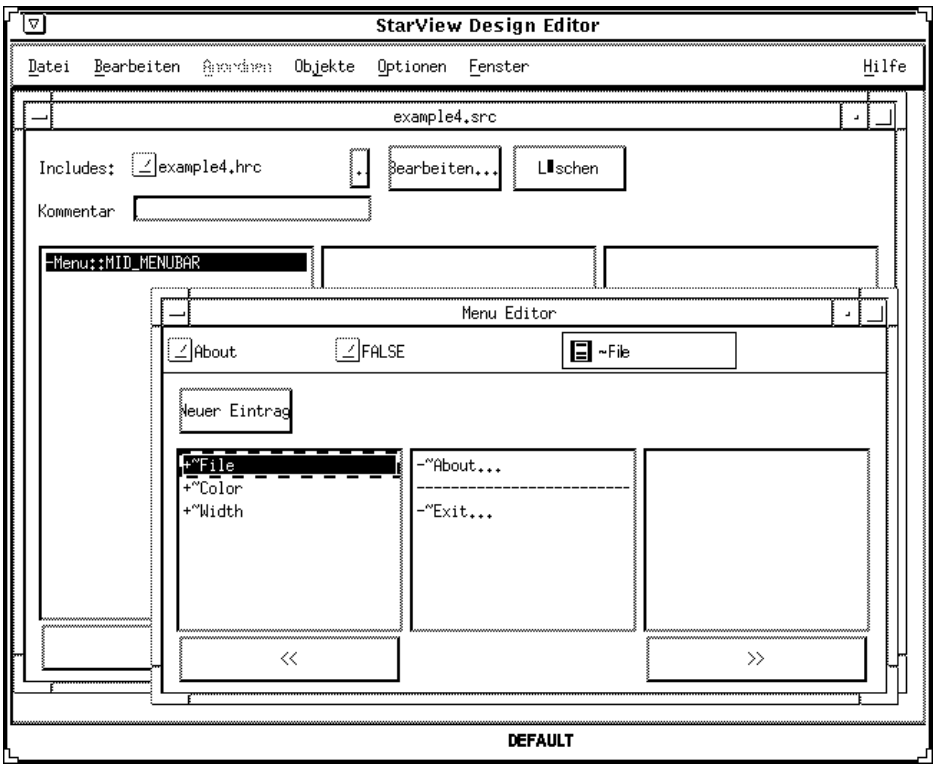


Abbildung 4.14: Bearbeitung des Menüeintrags *File* mit dem Design Editor

ter bearbeitet werden. In der Abbildung 4.14 sind für den Menüeintrag "File" die Folge-Menüeinträge "About..." und "Exit..." zu sehen. Diese werden durch eine Trennlinie (Separator) später im Menü getrennt sein.
Die Folge-Menüeinträge können direkt als Submenü an den jeweiligen Menüeintrag gehängt werden. Sie werden automatisch beim Laden des Menüs mitgeladen.

```
MenuItem
{
    Text       = "~File";
    Identifier = MID_FILE;
    SubMenu    = Menu
    {
        ItemList =
        {
            MenuItem
            {
                Text       = "~About...";
```

```
            Identifier = MID_ABOUT;
            About      = TRUE;
        };
        MenuItem
        {
            Separator = TRUE;
        };
        MenuItem
        {
            Text       = "~Exit...";
            Identifier = MID_QUIT;
        };
    };
};
};
```

Die Zuordnung dieser Folge-Menüeinträge zum jeweiligen Menüeintrag im Hauptprogramm (wie im vorigen Beispiel "PainterWithAboutBox") ist nicht mehr notwendig. Damit jedoch die benutzerdefinierten Funktionen (*handler*) an die Menüeinträge gebunden werden können, muß vorher der Zugriff auf die Menüeinträge gegeben sein. Dies erfolgt über die Funktion *menu.getPopupMenu*:

```
(* fetch menus to install Handler *)
let colorMenu = menu.getPopupMenu(menuBar PENCOLOR)
let widthMenu = menu.getPopupMenu(menuBar PENWIDTH)
let fileMenu = menu.getPopupMenu(menuBar FILE)

menu.pushSelectHdl(colorMenu link.new(colorSelect(mainWin colorMenu)))
menu.pushSelectHdl(widthMenu link.new(widthSelect(mainWin widthMenu)))
menu.pushSelectHdl(fileMenu link.new(fileSelect(mainWin fileMenu)))
```

**Die *AboutBox* aus Resourcen heraus geladen:**

In der Resource-Datei für die *AboutBox* stehen Angaben zu Größe, Position und Text. Über Schlüsselwörter werden die Eigenschaften der Box definiert.

```
ModalDialog RID_ABOUTBOX
{
    Size       = MAP_PIXEL( 200, 180 );
    Text       = "Paint";
    OutputSize = TRUE;
    Moveable   = TRUE;
    Hide       = TRUE;
    ...
};
```

Die Abbildung 4.15 zeigt die aufgeklappte Liste der Schlüsselwörter einer modalen Dialogbox. Bestimmte Eigenschaften können ausgewählt und verändert werden. Nicht veränderte Eigenschaften erhalten Standardwerte.

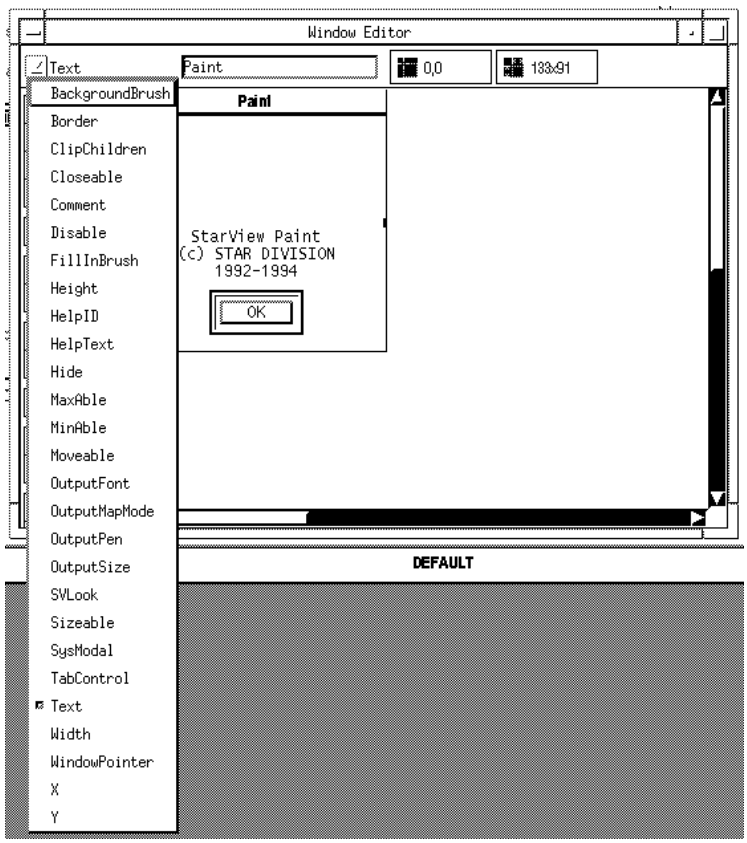


Abbildung 4.15: Liste der Schlüsselwörter der *AboutBox*

Der Ok-Knopf und der feste Text der *AboutBox* können als Sub-Resourcen der *AboutBox* in der Resource-Datei angegeben werden:

```
ModalDialog RID_ABOUTBOX
{
  ...
   OKButton RID_OK
   {
      PosSize    = MAP_PIXEL( 65, 135, 70, 25 );
      DefButton  = TRUE;
      TabStop    = TRUE;
   };
```

};

Die Abbildung 4.16 zeigt die Bearbeitung der *AboutBox* mit dem Design Editor.

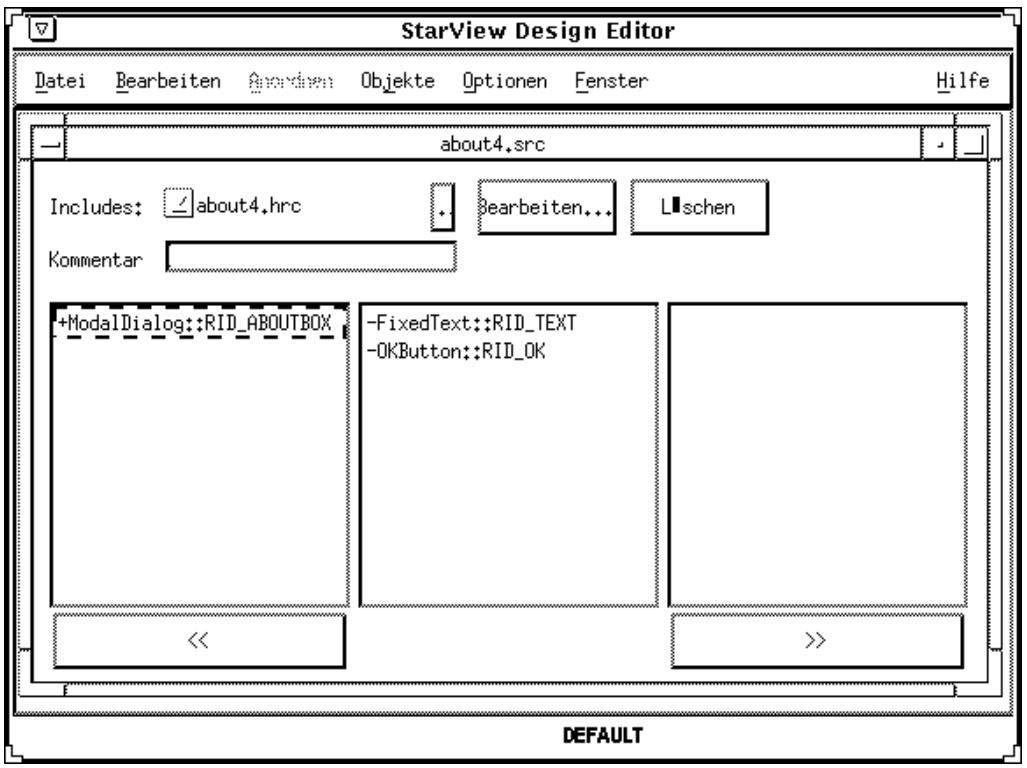


Abbildung 4.16: Bearbeitung der *AboutBox* mit dem Design Editor

In der Hauptfunktion des Beispiels werden die (übersetzten) Resourcen geladen. Der Pfad der (übersetzten) Resource-Datei ist dem Resource-Manager mitzuteilen. Mit der Funktion *menu.newBarfromRes* und der Angabe eines Identifiers wird die Menüleiste aus der Resource geladen.

*(* set resource manager to file painterWithResources.res *)*
*resource.setResManager(mainWin.win resMgr.new("svenv_examples/painterWithResources.res"))*

Damit sämtliche Sub-Resourcen der ModalDialogResource vom Resource-Manager aufgefunden und geladen werden können muß die Haupt-Resource der modalen Dialogbox bis zum erfolgten Laden der Sub-Resourcen gesperrt sein. Dieser automatische Vorgang wird mit Aufrufen der Funktion *resource.freeResource* nach dem Laden der Sub-Resourcen aufgehoben.

**let** *self = modalDialog.newFromRes(parent resId.new(ABOUTBOX))*

```
let text = fixedText.newFromRes(self resId.new(TEXT))
let but = okButton.newFromRes(self resId.new(OK))

resource.freeResource(self)
```

Bei Beenden der Anwendung wird auch automatisch die geladene Resource freigegeben. Gleichzeitig werden die Popupmenüs, die als Elemente der Menüleiste aus der Resource geladen wurden, entfernt.

## 4.4 Vor- und Nachteile der Wege

In den letzten drei Abschnitten wurden der implizite, explizite und interaktive Weg als Möglichkeit zur Entwicklung von graphischen Benutzeroberflächen vorgestellt. Der implizite Weg spricht über eine Zwischenschicht (*sveditenv*) ausgewählte Funktionen der StarView Bibliothek (*svenv*) an. Er ermöglicht ein schnelles Entwerfen von verschiedenen, eventuell ineinander geschachtelten Editoren. Allen Editoren ist eine einheitliche Präsentation und Dialogsteuerung gemein. Der explizite Weg verwendet direkt alle Funktionen der StarView Bibliothek (*svenv*). Zur Zeit dieser Arbeit sind nicht alle StarView Funktionen verfügbar, weil nur Funktionen verwendet werden sollen, die auf allen Plattformen funktionieren, auf denen das Tycoon-System portiert worden ist. Der interaktive Weg wurde anhand des Design Editor von StarView beschrieben. Leider ist der Design Editor und somit die Benutzung von Resourcen zur Zeit dieser Arbeit aus dem oben genannten Grund nicht möglich.
Der Vorteil beim impliziten Weg liegt in der Zwischenebene, den Modulen *editenv* und *sveditenv*. Eine Änderung der Implementation von einzelnen Funktionen aus diesen Modulen hat keinen Einfluß auf die Anwendungsprogramme, die auf dieser Zwischenebene aufsetzen. Der Nachteil dieses Weges ist, daß anwendungsspezifische Oberflächen nicht oder nur mit sehr viel Aufwand erstellt werden können.
Der Vorteil beim expliziten Weg ist die Verfügbarkeit aller Klassen und Funktionen der Bibliothek *svenv*. Hier kann eine Änderung von Implementationen von Funktionen großen Einfluß auf Anwendungsprogramme haben. Die Tabelle 4.2 faßt die obigen Punkte noch einmal zusammen.

| | **expliziter Weg** | **impliziter Weg** | **interaktiver Weg** |
|---|---|---|---|
| Aufwand bei anwendungsspezifischen Oberflächen | hoch | sehr hoch | reduziert |
| Aufwand bei Editoren | hoch | niedrig | reduziert |
| Möglichkeiten | viele | eingeschränkt | viele |
| Funktionen | (fast) alle | eingeschränkt | eingeschränkt |
| Portabilität zw. Fenstersystemen | mittel | hoch | mittel |
| Portabilität zw. Plattformen | mittel | hoch | mittel |
| Abhängigkeit der Anwendungsprogramme von verwendeten Funktionen | sehr hoch | niedrig | hoch |

Tabelle 4.2: Bewertung der drei Wege

# 5. Zusammenfassung und Ausblick

Dieses abschließende Kapitel schildert die Erfahrungen der Autorin und enthält eine kurze Zusammenfassung.

## 5.1 Erfahrungen mit StarView

Die Einarbeitung in eine so große und umfangreiche Klassenbibliothek wie StarView ist für Einsteiger sehr aufwendig. Hinzu kommt, daß StarView im Gegensatz zu Tycoon objektorientiert ist. Erlernte StarView-Konzepte müssen auf Tycoon-Konzepte abgebildet werden.
In dem StarView Handbuch [Busch et al. 93] sind ca. 20 einführende Beispiele zu verschiedenen Themen gegeben. Diese geben einen Einblick in die Leistungsfähigkeit dieser Software. Es ist nicht möglich, die Beispiele 1:1 in Tycoon zu übernehemen. Nicht alle Klassen und Funktionen stehen dem Tycoon-System zur Verfügung. Mit dem Referenzhandbuch, auf das im Handbuch häufiger verwiesen wird, ist das StarView online Hilfesystem (ein Hypertextsystem, mit StarView realisiert) gemeint. Diese Hilfe bietet, nach Auswahl einer Klasse, die zugehörige Klassendeklaration und eine Aufzählung der zugehörigen Funktionen. Zu jeder Funktion gibt es einen Verweis auf deren Deklaration und eine Beschreibung. Es existieren ebenfalls Querverweise auf die relevanten Basisklassen. Die Klassen werden aufgrund ihrer Vielzahl in einer alphabetisch sortierten Liste dargestellt. Diese Darstellungsart ist sehr unübersichtlich bezüglich der Hierarchie. Generell ist das Hilfesystem nicht sehr robust, z.B. führt ein einfacher Druckversuch innerhalb der Hilfe zu einem Programmabsturz. StarView liefert, neben den obligatorischen Klassen für GUIs, Klassen mit vorgefertigten Dialogen (z.B. Drucken, Dateiauswahl, Fontauswahl). So wie viele andere portable Klassenbibliotheken auch, bietet StarView keine Klasse zum einfachen Geometriemanagement[Schwarz 94b]. Das Oberflächendesign kann in speziellen Dateien (Resource-Dateien) beschrieben werden. Diese müssen jedoch in einer speziellen Syntax abgefaßt sein. Der Resourcecompiler übersetzt diese Datei in eine binäre Form. Bei der Verwendung der Resource-Datei in einer Anwendung ist darauf zu achten, daß sich die Datei im gleichen Verzeichnis wie das auszuführende Programm befindet. Findet das Programm die Datei nicht, wird noch eine Fehlermeldung ausgegeben und das Programm stürzt ab. Resource-Dateien in einem Texteditor zu erzeugen, bringt keine Vorteile gegenüber einer Programmierung mit der Klassenbibliothek, denn die Aktionen (*event handler*) auf die Ereignisse müssen vom Programmierer direkt mit der Klassenbibliothek geschrieben werden. Daher ist eine Erstellung von Resource-

Dateien mit dem Design Editor sinnvoller.

## 5.2 Erfahrungen mit dem Design Editor

Der Design Editor ist speziell für die Entwicklung der Oberfläche einer Anwendung vorgesehen und ist selber eine StarView-Anwendung. Die Oberflächenobjekte können über ein aufklappbares Menü oder eine separat angezeigte Palette ausgewählt werden. Das aufklappbare Menü zeigt alle verfügbaren Objekte an, Folge-Objekte werden als inaktiv markiert. Dies gewährt einen schnellen Überblick über die Objekthierarchie. Objekte über die Palette auszuwählen, erweist sich als schwierig, da die Icons der Palette viel zu klein und dadurch kaum lesbar sind (die Abbildung 2.1 zeigt die Palette zum Bearbeiten von Fenstern). Der Versuch, Objekte mit einem Hintergrundmuster zu versehen, führt zu einem Programmabsturz. Häufig werden die Objekte vom Editor nicht korrekt oder gar nicht dargestellt. Hier hilft nur, den Editor zu schließen und nochmals zu öffnen. Auch der Design Editor ist nicht sehr robust, häufigeres Sichern des Entwurfs empfiehlt sich. Der Design Editor hinterläßt den Eindruck, noch nicht in seiner endgültigen Version vorzuliegen.

## 5.3 Ausblick

Die Ablauf von GUI-Anwendungen erfolgt häufig nach folgendem Schema:

- Die statische Struktur wird festgelegt (Positionierung der graphischen Interaktionselemente.
- Die *event handler* werden definiert.
- Die Ereignisverarbeitung wird gestartet.

Nach dem Start der Ereignisverarbeitung sind keine Programmänderungen mehr möglich. Obwohl Tycoon die Möglichkeit der inkrementellen Übersetzung bietet, sind Änderungen zur Laufzeit der Anwendung nicht sichtbar. Die Anwendung muß verlassen werden, weil Änderungen sich nur bei neu gestarteter Ereignisverarbeitung auswirken.
Tycoon bietet die Möglichkeit, mehrere Kontrollflüsse (*threads of control*) auf demselben Zustand operieren zu lassen [Matthes, Schmidt 94]. So kann die Ereignisverarbeitung (*event loop*) in einem eigenständigen *thread*, z.B. dem *event loop thread*, gestartet werden. Parallel zu dem *event loop thread* läuft ein weiterer *thread* (z.B. der TL *top level*), in dem durch inkrementelles Übersetzen die Anwendung dynamisch modifiziert werden kann. Diese Änderungen sind sofort sicht- und testbar. Diese Art der interaktiven Erstellung einer graphischen Oberfläche ist gerade bei der Programmierung der *event handler* sehr viel komfortabler als mit herkömmlichen Werkzeugen (z.B. Design Editor oder SUN's *Developer's Graphical User Interface Design Editor* [Sun 92]) [Geisler 95]. Die Funktionalität der *event handler* kann iterativ entwickelt und getestet werden. Dies ist von Vorteil, denn bei der Erstellung einer graphischen Benutzeroberfläche stellen Reaktionen auf Ereignisse den größten Teil des Programmkodes dar.

Das Tycoon-System bietet noch keine graphische Unterstützung zur direkten Manipulation von GUI-Elementen an, wie z.B. die visuelle Programmierumgebung VisualBASIC [Maslo, Dittrich 93]. Zukünftige Arbeiten könnten sich mit dem Entwurf und Anbindung einer visuellen Programmierumgebung an persistente Objektsysteme (*VisualTycoon*) beschäftigen.

## 5.4 Zusammenfassung

Diese Arbeit stellt eine Anleitung zur Oberflächenprogrammierung mit Tycoon unter Verwendung der StarView C++ Klassenbibliothek dar. In Kapitel 2 werden eine Reihe von Entwicklungswerkzeugen vorgestellt, um graphische Benutzeroberflächen zu realisieren. Die Werkzeuge werden in dieser Arbeit nicht speziell auf ihre Handhabbarkeit untersucht. Die Grundlagen über StarView und die Verwendung der StarView Klassenbibliothek über das Tycoon-System liefert das Kapitel 3. Das Kapitel 4 liefert eine Reihe von Beispielen mit ausführlicher Beschreibung.

# A. Abbildung von C++-Klassen auf TL Module

Die objektorientierte Programmiersprache C++ wird zunehmend in der Industrie eingesetzt. Dies liegt zum einen daren, daß es sich um eine Sprache handelt, die mehrere Programmierstile zuläßt, und zum anderen, daß die viel verwendete Sprache C eine echte Teilsprache von C++ ist. Eine Idee der Objektorientierung, die Wiederverwendbarkeit von Kode, wird sehr betont. Hinzu kommt die Möglichkeit, die Klassen aus schon entwickelten Bibliotheken nach eigenen Wünschen zu modifizieren. Dies garantiert softwaretechnische Flexibilität. Ein Grundkonzept des Tycoon-Systems ist es, externe Bibliotheken zu nutzen [Matthes 93]. Eine externe Bibliothek (StarView) wurde an das Tycoon-System angebunden (siehe [Geisler 95]). Innerhalb von TL ist diese externe Bibliothek nutzbar.
In Tabelle A.1 werden die einander entsprechenden Sprachkonzepte von C++ und TL im Überblick dargestellt.

| C++ | TL |
|---|---|
| Konstante (*define*) | Wertbindung |
| Konstante (*enum*) | Export von Werten eines semi-abstrakten Typs (<:Int) |
| *class* | ADT mit Konstruktor- und Destruktorfunktion, Lese- und Schreibfunktionen |
| Klassenmethoden | Funktionen eines ADTs |
| Subtyppolymorphismus durch Vererbung | Subtyppolymorphismus durch einschränkende Typbindungen |

Tabelle A.1: Gegenüberstellung der Sprachkonzepte

StarView stellt zahlreiche C++ Klassen mit vielen Funktionen bereit. Das C++ Klassenkonzept wird in TL auf das Modulkonzept abgebildet. Die Namen der TL Module und Funktionen sind äquivalent zu den entsprechenden StarView Klassen und Funktionen. Eine Ausnahme bilden die Klassen *menu, menuBar* und *popupMenu*, die sich gegenseitig importieren. In TL sind diese Klassen in einem Modul namens *menu* zusammengefaßt, weil Module nicht rekursiv sein können.

In TL stehen die Modularisierungsmechanismen Modul, Modulschnittstelle und Bibliothek zur Verfügung. Um die Programme überschaubar zu halten, sollte für jede Klasse ein Modul geschrieben werden. Klassen werden in TL als abstrakter Datentyp dargestellt. Im Implementationsmodul wird das Sprachkonstrukt *Tuple* verwendet, um Klassen zu definieren.
Das folgende C++ Kodefragment zeigt die Definition einer Klasse.

```
// — class MyWin ————————————————————————-

class MyWin : public WorkWindow
{
private:
    USHORT          nColor;
    USHORT          nWidth;
    Point           aLastPoint;

public:
                MyWin();

...
};
```

Diese C++ Klasse stellt sich in TL wie folgt dar:

```
Let T =  Tuple
           win :workWindow.T
           var nColor  :Int
           var nWidth  :Int
           var aLastPoint  :point.T
         end
```

Ein Ausschnitt aus der dazugehörigen Schnittstelle:

```
T <: Tuple
       win :workWindow.T
     end
```

Die Tabelle A.2 zeigt Funktionsaufrufe in C++ aus StarView Beispielen und ihre äquivalenten Funktionsaufrufe in TL.
Beobachtung:
In C++ steht erst das Objekt und dann wird die Methode mit ihren Parametern angegeben.
*obj.maximize(...)*

In TL wird zuerst das Modul angegeben, dann die Funktion mit dem Parameter Objekt gefolgt von anderen Parametern.
*window.maximize(obj ...)*

| C++ | TL |
|---|---|
| *aMenuBar.InsertItem(MID_File, "~File");* | *menu.insertItem(aMenuBar mid3.File "~File")* |
| *aFileMenu.InsertItem(MID_ABOUT, "~About ...",MENU_APPEND, MIB_ABOUT);* | *menu.insertItem2(aFileMenu mid3.ABOUT "~About ..." menu.MENU_APPEND menu.MIB_ABOUT)* |
| *aColorMenu.InsertItem(MID_PENRED, "~Red", MENU_APPEND, MIB_CHECKABLE);* | *menu.insertItem2(aColorMenu mid3.PENRED "~Red" menu.MENU_APPEND menu.MIB_CHECKABLE)* |

Tabelle A.2: Funktionsaufrufe in C++ und TL

### Ein C++ Beispiel (HelloWorld)

Der folgende Programmkode erzeugt das in Abbildung 3.3 dargestellte Fenster. Das entsprechende TL Programm findet sich in Anhang B.1.

```
// — class MyApp ————————————————————————-

class MyApp : public Application
{
public:
   virtual void Main( int, char*[] );
};

// — class MyWin ————————————————————————-

class MyWin : public WorkWindow
{
public:
            MyWin( Window* pParent, WinBits nWinStyle ) :
               WorkWindow( pParent, nWinStyle )

   virtual void Paint( const Rectangle& );
};

// — MyWin::Paint() ———————————————————————-

void MyWin::Paint( const Rectangle& )
{
   DrawText( Point( 50, 50 ), "Keine Panik" );
}
```

```
// — MyApp::Main() ——————————————————————

void MyApp::Main( int, char*[] )
{
    MyWin aMainWin( NULL, WB_APP — WB_STDWORK );

    aMainWin.SetText( "StarView Hello World, mein erstes Fenster" );
    aMainWin.Show();

    Execute();
}

// — aMyApp ————————————————————————————

MyApp aMyApp;
```

# B. Die Beispiele "HelloWorld", "Painter", "PainterWithAboutBox", "PainterWithResources"

## B.1 Das Beispiel "HelloWorld"

```
module helloWorld
import
  application
  window :Window
  workWindow
  outputDevice
  point
  rectangle
  int

export

let main(argv :Array(String)) :Int =
  begin
    let aMainwin = workWindow.new2(window.nilT int.bitOr(window.WB_STDWORK
                                                         window.WB_APP))

    window.setPaint(aMainWin
      fun(pEvt :rectangle.T) :Ok
        begin
          outputDevice.drawText(aMainWin  point.new3(50 50)  "Keine Panik")
        end)

    window.setText(aMainWin "StarView Hello World, mein erstes Fenster")
```

```
    window.show(aMianWin)
    application.execute()
    0
  end
end;
```

## B.2 Das Beispiel "Painter"

```
module painter
(*
This module creates the application-window upon "aMainWin". Further the
menu with its entries and popup-menus with their entries are
initialized. The window is shown by the method "window.show(...)".
*)
import
  link
  application
  window :Window
  outputDevice
  menu
  workWindow
  systemWindow
  mouseEvent
  queryBox
  messBox
  pen
  color
  int
  point

export

let MID_PENCOLOR   = 10
let MID_PENRED     = 11
let MID_PENBLUE    = 12
let MID_PENYELLOW  = 13

let MID_PENWIDTH   = 20
let MID_PENSMALL   = 21
let MID_PENMEDIUM  = 22
let MID_PENTHICK   = 23


Let T = Tuple
          win              :workWindow.T
          var nColor       :Int
          var nWidth       :Int
          var aLastPoint   :point.T
        end
```

```
let colorSelect(win :T  pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
      if win.nColor != 0 then
        menu.checkItem2(pMenu win.nColor false)
      end
      win.nColor:= menu.getCurItemId(pMenu)
      menu.checkItem(pMenu win.nColor)
      let aPen = outputDevice.getPen(win.win)
        if win.nColor == MID_PENRED then
          pen.changeColor(aPen color.new2(color.COL_RED))
        elsif win.nColor == MID_PENBLUE then
          pen.changeColor(aPen color.new2(color.COL_BLUE))
        else
          pen.changeColor(aPen color.new2(color.COL_YELLOW))
        end
      outputDevice.changePen(win.win aPen)
    end

let widthSelect(win :T  pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
      if win.nWidth != 0 then
        menu.checkItem2(pMenu win.nWidth false)
      end
      win.nWidth:= menu.getCurItemId(pMenu)
      menu.checkItem(pMenu win.nWidth)
      let aPen = outputDevice.getPen(win.win)
        if win.nWidth == MID_PENSMALL then
          pen.changeWidth(aPen 1)
        elsif win.nWidth == MID_PENMEDIUM then
          pen.changeWidth(aPen 4)
        else
          pen.changeWidth(aPen 8)
        end
      outputDevice.changePen(win.win aPen)
    end

let myWin() =
begin
```

```
  let t = tuple
    let win = workWindow.new2(window.nilT
                              int.bitOr(window.WB_STDWORK window.WB_APP))
    let var nColor = 0
    let var nWidth = 0
    let var aLastPoint = point.nilT
  end

  systemWindow.setClose(t.win
    fun() :Bool
      begin
        let aBox = queryBox.new(t.win int.bitOr(window.WB_OK_CANCEL
                                                window.WB_DEF_OK)
                                "Exit Paint")
        if messBox.execute(aBox) == messBox.RET_OK then
          systemWindow.close(t.win)
        else
          false
        end
      end)

  window.setMouseButtonDown(t.win
    fun(mEvt :mouseEvent.T) :Ok
      begin
        window.captureMouse(t.win)
        t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
      end)

  window.setMouseButtonUp(t.win
    fun(mEvt :mouseEvent.T) :Ok
       window.releaseMouse(t.win))

  window.setMouseMove(t.win
    fun(mEvt :mouseEvent.T) :Ok
      if mouseEvent.isLeft(mEvt) then
        outputDevice.drawLine(t.win t.aLastPoint mouseEvent.getPosPixel(mEvt))
        t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
      end)

  t
end
```

```
let myMenu(aMainWin) :menu.Bar =
  begin
    let aColorMenu = menu.newPopup1()
    let aWidthMenu = menu.newPopup1()
    let menuBar = menu.newBar1()

    menu.insertItem(menuBar MID_PENCOLOR "~Color")
    menu.insertItem(menuBar MID_PENWIDTH "~Width")

    menu.changePopupMenu(menuBar MID_PENCOLOR aColorMenu)
    menu.changePopupMenu(menuBar MID_PENWIDTH aWidthMenu)

    menu.insertItem2(aColorMenu MID_PENRED "~Red"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aColorMenu MID_PENBLUE "~Blue"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aColorMenu MID_PENYELLOW "~Yellow"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)

    menu.insertItem2(aWidthMenu MID_PENSMALL "~Small"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aWidthMenu MID_PENMEDIUM "~Medium"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aWidthMenu MID_PENTHICK "~Large"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)

    menu.pushSelectHdl(aColorMenu
                       link.new(colorSelect(aMainWin aColorMenu)))
    menu.pushSelectHdl(aWidthMenu
                       link.new(widthSelect(aMainWin aWidthMenu)))
    menuBar
  end

let main(argv :Array(String)) :Int =
  begin
    let aMainWin = myWin()
    let aMenuBar = myMenu(aMainWin)
    application.changeAppMenu(aMenuBar)
    window.setText(aMainWin.win "StarView Paint")
    window.show(aMainWin.win)
    application.execute()
```

```
    0
  end
end;
```

## B.3 Das Beispiel "PainterWithAboutBox"

```
module painterWithAboutBox
(* Of the same kind as module "painter". *)
import
  link
  application
  window :Window
  workWindow
  menu
  systemWindow
  mouseEvent
  queryBox
  messBox
  outputDevice
  pen
  color
  int
  link
  modalDialog
  application
  point
  polygon
  brush
  fixedText
  modalDialog
  okButton
  rectangle
  size

export

let PENCOLOR   = 10
let PENRED     = 11
let PENBLUE    = 12
let PENYELLOW = 13

let PENWIDTH   = 20
let PENSMALL   = 21
let PENMEDIUM = 22
let PENTHICK   = 23

let FILE       = 30
```

```
let QUIT      = 31
let ABOUT     = 32

Let T = Tuple
          win              :workWindow.T
          var nColor       :Int
          var nWidth       :Int
          var aLastPoint   :point.T
        end

(*
The function "newAboutBox" creates an about-box. About-box is of type modalDialog. That
means that the about-box is active as long as it is not finished. In the about-box finds a text, a
button and three coloured geometrical figures. The about-box is finished by clicking on the OK-
button.
*)

let newAboutBox(parent :window.T) :modalDialog.T =
   begin

     let modDiag = modalDialog.new2(parent window.WB_STDMODAL)
     let aText = fixedText.new1(modDiag)
     let aBut =
        okButton.new2(modDiag int.bitOr(window.WB_DEFBUTTON
                                        window.WB_TABSTOP))

     window.setText(modDiag "Paint")
     window.changeOutputSizePixel(modDiag size.new3(200 180))
     window.setText(aText "StarView Paint\n (c) STAR DIVISION\n 1992-1994")

     window.changePosPixel(aText point.new3(25 80))
     window.changeSizePixel(aText size.new3(150 50))
     window.show(aText)

     window.changePosPixel(aBut point.new3(65 135))
     window.changeSizePixel(aBut size.new3(70 25))
     window.show(aBut)

     let paint(rect :rectangle.T) :Ok =
       begin
         let aOldBrush =
           outputDevice.changeFillInBrush(modDiag brush.new4(color.new2(color.COL_RED)))
```

```
        outputDevice.drawRect(modDiag rectangle.new3(point.new3(20 10)
                                                     point.new3(80 50)))

        outputDevice.changeFillInBrush(modDiag
                                       brush.new4(color.new2(color.COL_YELLOW)))
        outputDevice.drawEllipse(modDiag rectangle.new3(point.new3(60 20)
                                                        point.new3(130 60)))


        outputDevice.changeFillInBrush(modDiag
                                       brush.new4(color.new2(color.COL_GREEN)))
        let poly = polygon.new2(3)
        polygon.changePoint(poly point.new3(110 70) 0)
        polygon.changePoint(poly point.new3(140 30) 1)
        polygon.changePoint(poly point.new3(170 70) 2)
        outputDevice.drawPolygon(modDiag poly)

        outputDevice.changeFillInBrush(modDiag aOldBrush)
      end

      window.setPaint(modDiag paint)
      modDiag
    end


let colorSelect(win :T pMenu :menu.T) :link.Handler =
  fun() :Ok
      begin
        if win.nColor != 0 then
          menu.checkItem2(pMenu win.nColor false)
        end
        win.nColor:= menu.getCurItemId(pMenu)
        menu.checkItem(pMenu win.nColor)
        let aPen = outputDevice.getPen(win.win)
          if win.nColor == PENRED then
            pen.changeColor(aPen color.new2(color.COL_RED))
          elsif win.nColor == PENBLUE then
            pen.changeColor(aPen color.new2(color.COL_BLUE))
          else
            pen.changeColor(aPen color.new2(color.COL_YELLOW))
          end
        outputDevice.changePen(win.win aPen)
```

```
    end

let widthSelect(win :T  pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
      if win.nWidth != 0 then
        menu.checkItem2(pMenu win.nWidth false)
      end
      win.nWidth:= menu.getCurItemId(pMenu)
      menu.checkItem(pMenu win.nWidth)
      let aPen = outputDevice.getPen(win.win)
      if win.nWidth == PENSMALL then
        pen.changeWidth(aPen 1)
      elsif win.nWidth == PENMEDIUM then
        pen.changeWidth(aPen 4)
      else
        pen.changeWidth(aPen 8)
      end
      outputDevice.changePen(win.win aPen)
    end

let fileSelect(win :T pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
      if menu.getCurItemId(pMenu) == ABOUT then
        modalDialog.execute(about3.newAboutBox(win.win))
        ok
      else
        let aBox = queryBox.new(win.win int.bitOr(window.WB_OK_CANCEL
                                                  window.WB_DEF_OK)
                                "Exit Paint")
        if messBox.execute(aBox) == messBox.RET_OK then
          systemWindow.close(win.win)
        else
          false
        end
      end
    end

let myWin() :T =
  begin
    let t = tuple
```

```
        let win = workWindow.new2(window.nilT int.bitOr(window.WB_STDWORK
                                                        window.WB_APP))
        let var nColor = 0
        let var nWidth = 0
        let var aLastPoint = point.nilT
      end


  let mouseButtonDown(mEvt :mouseEvent.T) :Ok =
    begin
      window.captureMouse(t.win)
      t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
    end
  window.setMouseButtonDown(t.win mouseButtonDown)

  let mouseButtonUp(mEvt :mouseEvent.T) :Ok =
     window.releaseMouse(t.win)
  window.setMouseButtonUp(t.win mouseButtonUp)

  let mouseMove(mEvt :mouseEvent.T) :Ok =
    if mouseEvent.isLeft(mEvt) then
      outputDevice.drawLine(t.win t.aLastPoint mouseEvent.getPosPixel(mEvt))
      t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
    end
  window.setMouseMove(t.win mouseMove)

  t
 end


let myMenu(aMainWin :T) :menu.Bar =
  begin
    let aFileMenu = menu.newPopup1()
    let aColorMenu = menu.newPopup1()
    let aWidthMenu = menu.newPopup1()
    let menuBar = menu.newBar1()

    menu.insertItem(menuBar FILE "~File")
    menu.insertItem(menuBar PENCOLOR "~Color")
    menu.insertItem(menuBar PENWIDTH "~Width")

    menu.changePopupMenu(menuBar FILE aFileMenu)
```

```
    menu.changePopupMenu(menuBar PENCOLOR aColorMenu)
    menu.changePopupMenu(menuBar PENWIDTH aWidthMenu)

    menu.insertItem2(aFileMenu ABOUT "~About..."
                     menu.MENU_APPEND menu.MIB_ABOUT)
    menu.insertSeparator(aFileMenu)
    menu.insertItem(aFileMenu QUIT "~Exit...")

    menu.insertItem2(aColorMenu PENRED "~Red"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aColorMenu PENBLUE "~Blue"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aColorMenu PENYELLOW "~Yellow"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)

    menu.insertItem2(aWidthMenu PENSMALL "~Small"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aWidthMenu PENMEDIUM "~Medium"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)
    menu.insertItem2(aWidthMenu PENTHICK "~Large"
                     menu.MENU_APPEND menu.MIB_CHECKABLE)

    menu.pushSelectHdl(aFileMenu
                       link.new(fileSelect(aMainWin aFileMenu)))
    menu.pushSelectHdl(aColorMenu
                       link.new(colorSelect(aMainWin aColorMenu)))
    menu.pushSelectHdl(aWidthMenu
                       link.new(widthSelect(aMainWin aWidthMenu)))
    menuBar
end

let main(argv :Array(String)) :Int =
  begin
    let aMainWin = myWin()
    let aMenuBar = myMenu(aMainWin)
    application.changeAppMenu(aMenuBar)
    window.setText(aMainWin.win "StarView Paint")
    window.show(aMainWin.win)
    application.execute()
    0
  end
end;
```

## B.4 Das Beispiel "PainterWithResources"

```
module painterWithResources
(*
Loads the application window from the resource.
*)
import
  link
  resMgr
  resId
  resource
  application
  window
  workWindow
  menu
  systemWindow
  mouseEvent
  queryBox
  messBox
  outputDevice
  pen
  color
  int
  modalDialog
  point
  polygon
  brush
  fixedText
  modalDialog
  okButton
  rectangle

export

let PENCOLOR    = 10
let PENRED      = 11
let PENBLUE     = 12
let PENYELLOW = 13

let PENWIDTH   = 20
let PENSMALL    = 21
let PENMEDIUM = 22
let PENTHICK    = 23
```

```
let FILE      = 30
let QUIT      = 31
let ABOUT     = 32

let ABOUTBOX  = 1100
let TEXT      = 1
let OK        = 2

let MENUBAR = 1000

Let T = Tuple
          win              :workWindow.T
          var nColor       :Int
          var nWidth       :Int
          var aLastPoint   :point.T
        end

(*
The function "newAboutBox" creates an about-box. About-box is of type modalDialog. That
means that the about-box is active as long as it is not finished. In the about-box finds a text, a but-
ton and three coloured geometrical figures. The about-box is finished by clicking on the OK-
button. Unlike the function "newAboutBox" in the example above this function works with resources.
*)
let newAboutBox(parent :window.T) :modalDialog.T =
  begin
    let self = modalDialog.newFromRes(parent resId.new(ABOUTBOX))
    let text = fixedText.newFromRes(self resId.new(TEXT))
    let but = okButton.newFromRes(self resId.new(OK))

    resource.freeResource(self)

    window.setPaint(self
      fun(rect :rectangle.T) :Ok
        begin
          let oldBrush =
              outputDevice.changeFillInBrush(self brush.new4(color.new2(color.COL_RED)))

          outputDevice.drawRect(self rectangle.new3(point.new3(20 10)
                                                    point.new3(80 50)))
          outputDevice.changeFillInBrush(self brush.new4(color.new2(color.COL_BLUE)))
          outputDevice.drawEllipse(self rectangle.new3(point.new3(60 20)
```

```
                                                point.new3(130 60)))
        outputDevice.changeFillInBrush(self
                                              brush.new4(color.new2(color.COL_GREEN)))
        let poly = polygon.new2(3)
        polygon.changePoint(poly point.new3(110 70) 0)
        polygon.changePoint(poly point.new3(140 30) 1)
        polygon.changePoint(poly point.new3(170 70) 2)
        outputDevice.drawPolygon(self poly)
        outputDevice.changeFillInBrush(self oldBrush)
       end)
    self
  end

let colorSelect(win :T pMenu :menu.T) :link.Handler =
  fun() :Ok
     begin
      if win.nColor != 0 then
        menu.checkItem2(pMenu win.nColor false)
      end
      win.nColor:= menu.getCurItemId(pMenu)
      menu.checkItem(pMenu win.nColor)
      let aPen = outputDevice.getPen(win.win)
       if win.nColor == PENRED then
         pen.changeColor(aPen color.new2(color.COL_RED))
       elsif win.nColor == PENBLUE then
         pen.changeColor(aPen color.new2(color.COL_BLUE))
       else
         pen.changeColor(aPen color.new2(color.COL_YELLOW))
       end
     outputDevice.changePen(win.win aPen)
     end

let widthSelect(win :T  pMenu :menu.T) :link.Handler =
    fun() :Ok
      begin
       if win.nWidth != 0 then
         menu.checkItem2(pMenu win.nWidth false)
       end
       win.nWidth:= menu.getCurItemId(pMenu)
       menu.checkItem(pMenu win.nWidth)
       let aPen = outputDevice.getPen(win.win)
        if win.nWidth == PENSMALL then
```

```
        pen.changeWidth(aPen 1)
      elsif win.nWidth == PENMEDIUM then
        pen.changeWidth(aPen 4)
      else
        pen.changeWidth(aPen 8)
      end
     outputDevice.changePen(win.win aPen)
    end

let fileSelect(win :T pMenu :menu.T) :link.Handler =
  fun() :Ok
    begin
     if menu.getCurItemId(pMenu) == ABOUT then
       modalDialog.execute(newAboutBox(win.win))
       ok
     else
       let aBox = queryBox.new(win.win int.bitOr(window.WB_OK_CANCEL
                                                 window.WB_DEF_OK)
                               "Exit Paint")
       if messBox.execute(aBox) == messBox.RET_OK then
         systemWindow.close(win.win)
       else
         false
       end
     end
    end

let myWin() :T =
 begin
   let t = tuple
       let win = workWindow.new2(window.nilT int.bitOr(window.WB_STDWORK
                                                       window.WB_APP))
       let var nColor = 0
       let var nWidth = 0
       let var aLastPoint = point.nilT
      end


   let mouseButtonDown(mEvt :mouseEvent.T) :Ok =
     begin
       window.captureMouse(t.win)
       t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
```

```
    end
  window.setMouseButtonDown(t.win mouseButtonDown)

  let mouseButtonUp(mEvt :mouseEvent.T) :Ok =
    window.releaseMouse(t.win)
  window.setMouseButtonUp(t.win mouseButtonUp)

  let mouseMove(mEvt :mouseEvent.T) :Ok =
    if mouseEvent.isLeft(mEvt) then
      outputDevice.drawLine(t.win t.aLastPoint mouseEvent.getPosPixel(mEvt))
      t.aLastPoint:= mouseEvent.getPosPixel(mEvt)
    end
  window.setMouseMove(t.win mouseMove)

  t
 end

let myMenu(aMainWin :T) :menu.Bar =
 begin
  (* set resource manager to file painterWithResources.res *)
  resource.setResManager(aMainWin.win
                         resMgr.new("svenv_examples/painterWithResources.res"))
  (* fetch menuBar from Resource *)
  let menuBar = menu.newBarfromRes(resId.new(MENUBAR))

  (* fetch menus to install Handler *)
  let colorMenu = menu.getPopupMenu(menuBar PENCOLOR)
  let widthMenu = menu.getPopupMenu(menuBar PENWIDTH)
  let fileMenu = menu.getPopupMenu(menuBar FILE)

  menu.pushSelectHdl(colorMenu
                     link.new(colorSelect(aMainWin colorMenu)))
  menu.pushSelectHdl(widthMenu
                     link.new(widthSelect(aMainWin widthMenu)))
  menu.pushSelectHdl(fileMenu
                     link.new(fileSelect(aMainWin fileMenu)))
  menuBar
 end

let main(argv :Array(String)) :Int =
 begin
  let aMainWin = myWin()
```

```
    let aMenuBar = myMenu(aMainWin)
    application.changeAppMenu(aMenuBar)
    window.setText(aMainWin.win "StarView Paint")
    window.show(aMainWin.win)
    application.execute()
    0
  end
end;

/*************************************************************
*  painterWithResources.src
*  (c) 1992-1994 STAR DIVISION
*************************************************************/

#include "painterWithResources.hrc"

Menu MID_MENUBAR
{
    ItemList =
    {
        MenuItem
        {
            Text       = "~File";
            Identifier = MID_FILE;
            SubMenu    = Menu
            {
                ItemList =
                {
                    MenuItem
                    {
                        Text       = "~About...";
                        Identifier = MID_ABOUT;
                        About      = TRUE;
                    };
                    MenuItem
                    {
                        Separator = TRUE;
                    };
                    MenuItem
                    {
                        Text       = "~Exit...";
                        Identifier = MID_QUIT;
                    };
```

```
        };
    };
};

MenuItem
{
    Text       = "~Color";
    Identifier = MID_PENCOLOR;
    SubMenu    = Menu
    {
        ItemList =
        {
            MenuItem
            {
                Text       = "~Red";
                Identifier = MID_PENRED;
                Checkable  = TRUE;
            };
            MenuItem
            {
                Text       = "~Blue";
                Identifier = MID_PENBLUE;
                Checkable  = TRUE;
            };
            MenuItem
            {
                Text       = "~Yellow";
                Identifier = MID_PENYELLOW;
                Checkable  = TRUE;
            };
        };
    };
};

MenuItem
{
    Text       = "~Width";
    Identifier = MID_PENWIDTH ;
    SubMenu    = Menu
    {
        ItemList =
        {
```

```
                MenuItem
                {
                    Text       = "~Small";
                    Identifier = MID_PENSMALL;
                    Checkable  = TRUE;
                };
                MenuItem
                {
                    Text       = "~Medium";
                    Identifier = MID_PENMEDIUM;
                    Checkable  = TRUE;
                };
                MenuItem
                {
                    Text       = "~Large";
                    Identifier = MID_PENTHICK;
                    Checkable  = TRUE;
                };
            };
        };
    };
};
};


/*************************************************************************
*  painterWithResources.hrc
*  (c) 1992-1994 STAR DIVISION
*************************************************************************/

#define MID_MENUBAR   1000
#define MID_PENCOLOR  10
#define MID_PENRED    11
#define MID_PENBLUE   12
#define MID_PENYELLOW 13
#define MID_PENWIDTH  20
#define MID_PENSMALL  21
#define MID_PENMEDIUM 22
#define MID_PENTHICK  23
#define MID_FILE      30
#define MID_QUIT      31
#define MID_ABOUT     32
```

```
/*************************************************************************
*  painterWithResourcesAbout.src
*  (c) 1992-1994 STAR DIVISION
*************************************************************************/

#include "painterWithResourcesAbout.hrc"

ModalDialog RID_ABOUTBOX
{
   Size       = MAP_PIXEL( 200, 180 );
   Text       = "Paint";
   OutputSize = TRUE;
   Moveable   = TRUE;
   Hide       = TRUE;

   FixedText RID_TEXT
   {
      PosSize  = MAP_PIXEL( 25, 80, 150, 50 );
      Text     = "StarView Paint\n(c) STAR DIVISION\n1992-1994";
      Center   = TRUE;
   };

   OKButton RID_OK
   {
      PosSize   = MAP_PIXEL( 65, 135, 70, 25 );
      DefButton = TRUE;
      TabStop   = TRUE;
   };
};


/*************************************************************************
*  painterWithResourcesAbout.hrc
*  (c) 1992-1994 STAR DIVISION
*************************************************************************/

#define RID_ABOUTBOX  1100
#define RID_TEXT      1
#define RID_OK        2
```

# Literaturverzeichnis


Borland 94 *Borland C++ 4.0: Das Buch.* 1994.

Busch et al. 93 Busch, A., Kuehnel, T. und Jahnke, A. *StarView 2.0.* Star Division, Hamburg, 1993.

Fähnrich et al. 92 Fähnrich, K.-P., Janssen, C. und Groh, G. *Entwicklungswerkzeuge für graphische Benutzerschnittstellen.* Computer Magazin, 1992, Nr. 2, S. 6-13.

Fran 87 Franke H.W. *Leonardo 2000 - Kunst im Zeitalter des Computers.* Fischer Taschenbuch Verlag, Frankfurt (1978), 1987

Geisler 95 Geisler, A. *Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1995.

HüSha 93 Hüskes, R. und Shahrbabaki, K. *Normierter Luxus, GUIs in Theorie und Praxis.* c't Magazin für Computertechnik, 1993, Nr. 9, S. 68-72.

Isau 93a Isau, R. *Was ihr wollt, Lösungsansätze zur plattformübergreifenden Programmierung.* c't Magazin für Computertechnik, 1993, Nr. 10, S. 68-74.

Isau 93b Isau, R. *Der Widerspänstigen Zähmung, Entwicklungswerkzeuge zur plattformunabhängigen Programmierung.* c't Magazin für Computertechnik, 1993, Nr. 11, S. 150-157.

Kröm 92 Krömker, D., *Visualisierungssysteme.* Springer-Verlag Berlin Heidelberg New-York 1992

Markus 92 Markus, Chr. *Untersuchung verschiedener Methoden zur Entwicklung von Programmen mit grafischer Benutzerschnittstelle.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1992.

Maslo, Dittrich 93 Maslo, P. und Dittrich, S. *Das große Buch zu VisualBASIC 3.0 für Windows.* Data Becker, Düsseldorf, 1993.

Matthes 93 Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

Matthes, Schmidt 94 Matthes, F. und Schmidt, J.W. *Persistent Threads In: Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB.* Santiago, Chile, September 1994, S. 403-414.

Meyer 88 Meyer, B. *Object-oriented Software Construction.* Prentice Hall International Inc., London, 1988.

MoSch 92 Monjau, D. und Schulze, S. *Objektorientierte Programmierung, Ein einführendes Lehrbuch mit Beispielen in Modula-2.* Vieweg Braunschweig, Wiesbaden, 1992.

Müßig 94 Müßig, S. *Beiträge zur typsicheren generischen Datenvisualisierung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

SchwPin 95 Schweizer, P. und Dr. Pinnow, K.-M. *Menü für Zwei, Dialog zwischen Programmierer und Grafiker.* c't Magazin für Computertechnik, 1995, Nr. 2, S. 244-248.

Schwarz 94a Schwarz, S. *Bibliomanen mit Motif, C++-Klassenbibliotheken: DiaViews++ 1.0, GRITplus 2.1, Open Interface Library 4.0, XVT++ 2.0.* iX 1994, Nr. 1, S. 32-43.

Schwarz 94b Schwarz, S. *Ausblick für Bibliomanen, C++-Klassenbibliotheken: StarView 2.1, Zinc 3.6.* iX 1994, Nr. 5, S. 82-90.

Sun 92 Sun Microsystems, Inc. *OpenWindows Developer's Guide 2.0 User's Guide.* Mountain View, 1992.

SunSoft 92 SunSoft, Inc. *The NeWS Toolkit 3.1 Reference Manual.* SunSoft, Inc., Mountain View, 1992.

Wegener 95 Wegener, H. *Die Erfindung des Rades, Wiederverwendbare Komponenten für die Benutzerschnittstelle.* iX 1995, Nr. 3, S. 34-49.