Technische Universität München
Institut für Informatik

Lehrstuhl für Ingenieuranwendungen in der Informatik
und numerische Programmierung

Einführung in InterViews

Stefan Mayer
Ohmstraße 8a
8440 Straubing

e-mail: mayer@informatik.tu–muenchen.de

20. Juli 1992

Vorwort

Mit der zunehmenden Verbreitung von Graphikbildschirmen gewannen in den letzten Jahren auch graphische Benutzeroberflächen immer stärker an Bedeutung. Sie ermöglichen es dem Anfänger ebenso wie dem fortgeschrittenen Benutzer von Rechnersystemen, sich in der Bedienung komplexer Programmsysteme leichter zurechtzufinden. Dabei ist natürlich ein möglichst einheitliches Erscheinungsbild von Anwendungen mit graphischen Oberflächen wünschenswert. Im Bereich der UNIX-Workstations wurde mit dem X-Window System ein weithin anerkannter Standard geschaffen. Allerdings erwies sich die Programmierung von X als schwer und unhandlich. Daher wurde an der Stanford University damit begonnen, ein auf X basierendes Toolkit für graphische Oberflächen zu entwickeln. Da solche Oberflächen einen "interactive view" auf Programmdaten bieten sollen, wurde das Toolkit "InterViews" getauft. InterViews wurde objektorientiert in C++ implementiert und bietet für alle bekannten Elemente graphischer Oberflächen wie z.B. Buttons eigene Klassen an. Dadurch wird die Programmierung erheblich vereinfacht.

Leider gab es zu InterViews bisher noch keine ausreichende Dokumentation. Die einzige einigermaßen brauchbare, für Anfänger aber völlig ungeeignete Beschreibung bietet das in Teilen unvollständige oder veraltete InterViews Reference Manual [LIN 91]. Auch ich selbst verbrachte beim Einstieg in die InterViews-Programmierung viele Stunden mit ziellosem Nachschlagen in diesem Manual. Erst das Studium der wenigen verfügbaren Beispielprogramme ermöglichte es mir allmählich, erste eigene Anwendungen zu schreiben. Aus dieser Situation heraus entstand die Idee, ein einleitendes Tutorial zu schreiben. Es soll anhand einfacher Beispiele die Grundlagen der InterViews-Programmierung erklären. Wer die Konzepte einmal verstanden hat, wird immer wieder gerne auf diesen mächtigen Werkzeugkasten zurückgreifen.

Diese Einführung bezieht sich auf den Stand Anfang 1992 (InterViews Version 3.0). Die hier beschriebenen Grundlagen sollten sich aber auch in Zukunft nicht wesentlich ändern, so daß dieses kleine Buch hoffentlich vielen Interessierten den Einstieg erleichtern kann.

Abschließend möchte ich mich bei allen bedanken, die mir bei meiner Arbeit geholfen haben. Besonderer Dank gebührt dabei Michael Arbesmeier, der mir immer wieder wertvolle Hinweise geben konnte.

Trotz aller Sorgfalt enthält dieses Buch sicher noch etliche Fehler, so daß ich für Verbesserungsvorschläge und andere Anregungen jederzeit dankbar bin.

Straubing, 20. Juli 1992 Stefan Mayer.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel möchte ich zunächst klarstellen, welche Kenntnisse der Leser mitbringen sollte, um dieses Tutorial verstehen zu können. Danach werden ein paar Grundlagen für den Umgang mit InterViews beschrieben, die abschließend anhand eines Beispiels eingeübt werden.

1.1 Anforderungen an den Leser

InterViews selbst ist in C++ geschrieben und steht dem Anwender in Form einer C++-Bibliothek zur Verfügung. Da InterViews-Applikationen demzufolge in C++ programmiert werden, setze ich die Kenntnis dieser Programmiersprache voraus. Besonders wichtig sind dabei die Ableitung von Klassen und die Vererbungsmechanismen.

Selbstverständlich soll der Leser auch mit dem System umgehen können, auf dem er InterViews benützen will. Üblicherweise wird dies wohl eine UNIX-Workstation sein.

Außerdem ist ein wenig Erfahrung im Umgang mit graphischen Oberflächen von Vorteil. Begriffe wie Fenster oder Maus sollten für Sie keine Fremdwörter mehr sein. Falls Sie auf diesem Gebiet noch etwas unsicher sind, machen Sie sich bitte mit Ihrem System vertraut.

1.2 Grundlegende Definitionen

In diesem Unterabschnitt werden zwei Typen und eine Klasse vorgestellt, die eine zentrale Bedeutung in InterViews haben.

boolean:

Der Typ boolean ist in einem Programm verfügbar, wenn ein beliebiges InterViews-Headerfile eingebunden wird. Er umfaßt die Konstanten *true* und *false* und ist ebenso aufzufassen wie in herkömmlichen Programmiersprachen (z.B. Pascal).

Coord:

Ebenso wie boolean steht dieser Typ sofort nach dem Einbinden eines InterViews-Headerfiles zur Verfügung. Viele Positionsangaben, die die Ausgabe am Bildschirm betreffen, werden in Coords ausgedrückt. Der Typ Coord ist in der gegenwärtigen InterViews-Version als float definiert.

Resource:

Diese Klasse bildet die Basisklasse für viele InterViews-Klassen wie z.B. Color, Font, Brush oder Glyph, die später erklärt werden. Objekte dieser Klassen haben gemeinsam, daß sie häufig an mehreren Stellen unabhängig voneinander benötigt werden. Ein gutes Beispiel dafür sind Farben, die durch Objekte der Klasse Color repräsentiert werden. So wird für eine bestimmte Anzahl schwarzer Linien entsprechend oft die Farbe Schwarz verwendet. Es wäre Speicherplatzverschwendung, zehnmal ein Color-Objekt zu erzeugen. Wesentlich sinnvoller ist es, dasselbe Objekt an den verschiedenen Anwendungsstellen einzusetzen. Solche Objekte werden als "shared objects" bezeichnet. Shared objects müssen aber irgendwo verwaltet werden. So sollte beispielsweise ein nicht länger benötigtes Objekt, d.h. ein Objekt, das keine Anwendungsstellen mehr hat, automatisch gelöscht werden. Diese Leistungen erbringt die Basisklasse Resource. Deren Deklaration mit den für uns wichtigen Memberfunktionen sieht wie folgt aus:

```
class Resource
  {
    public:
      void ref() const;
      void unref() const;
      ...
  };
```

Jedes Objekt einer Klasse, die von Resource abgeleitet ist, enthält einen sogenannten Referenzzähler, der mit Null initialisiert ist. Dieser Referenzzähler gibt an, wie oft ein shared object im Moment verwendet wird. ref erhöht den Zähler um eins, mittels unref wird er dagegen um eins erniedrigt. Erreicht der Referenzzähler durch einen unref–Aufruf wieder den Wert Null, so wird das Objekt automatisch gelöscht. *Shared Objects dürfen niemals mit delete zerstört werden! Außerdem werden shared objects nur mittels new erzeugt und dürfen demzufolge nicht als automatische Variablen verwendet werden.* Der Grund dafür liegt darin, daß durch einen expliziten delete-Aufruf oder die selbständige Löschung automatischer Variablen der Verwaltungsmechanismus gestört und zu Speicherfehlern führen würde.

Eine weitere Regel ist noch wichtig: Wird ein shared object in einem Programmstück P erzeugt und lediglich an eine Funktion F als Argument übergeben, so braucht das Objekt in P nicht referenziert bzw. dereferenziert werden. Diese Aufgabe muß nach einer InterViews-Konvention F übernehmen. Jede Routine, die ein shared object (genauer: einen Zeiger auf ein shared object) als Argument hat, muß dieses zu Beginn des Rumpfs mittels ref referenzieren und am Ende des Rumpfes durch unref dereferenzieren.

Wir wollen uns diese Sachverhalte nochmals anhand dreier Beispiele verdeutlichen. Die genaue Bedeutung der verwendeten Klassen und Memberfunktionen ist an dieser Stelle nicht wichtig. Sie sollten lediglich das Prinzip verstehen.

Das folgende Programmfragment zeigt die normale Nutzung eines shared objects:

```
Color *black = new Color(0.0,0.0,0.0);
black->ref();   // Referenzzaehler erhoehen
...             // black verwenden
black->unref(); // Referenzzaehler erniedrigen
```

Die Anwendung des Color-Objekts wird mittels ref bzw. unref geklammert. Dieses Vorgehen ist aber bei InterViews-Objekten die Ausnahme. Solche shared objects werden meist an andere Routinen übergeben, wodurch die explizite Referenzierung bzw. Dereferenzierung entfällt:

```
Color *black = new Color(0.0,0.0,0.0);
canvas->fill(black); // Anwendungsstelle wird nicht
                     // durch ref und unref geklammert
...
```

In solchen Fällen müssen die Routinen, die Zeiger auf shared objects als Argumente erhalten, die Referenzierung bzw. Dereferenzierung übernehmen. Die Implementation der Canvas-Memberfunktion fill erhält daher in etwa folgenden Rumpf:

```
void Canvas::fill(Color *col)
  {
    col->ref();   // col referenzieren
    ...
    col->unref(); // col dereferenzieren
  }
```

1.3 Beispiel

Nachdem wir uns bisher ausschließlich mit theoretischen Fragen beschäftigt haben, soll jetzt ein erstes InterViews-Programm vorgestellt werden. Das Programm bringt ein kleines Fenster mit fester Größe auf den Bildschirm, das den Text "hallo!" enthält. Der zugehörige C++-Code sieht folgendermaßen aus:

```
#include <InterViews/session.h>
#include <InterViews/display.h>
#include <InterViews/style.h>
#include <InterViews/window.h>
#include <InterViews/background.h>
#include <InterViews/label.h>

int main( int argc, char **argv)
  {
    Session *session = new Session("Example",argc,argv);
    Style *style = session->default_display()->style();

    return session->run_window(
                new ApplicationWindow(
                   new Background(
                      new Label("hello!",style->font(),
                                         style->foreground()
                                )，
                      style->background()
                                 )
                                     )
                              );
  }
```

Wenn Sie das Programm jetzt sofort ausprobieren wollen, dann folgen Sie bitte den Ratschlägen im Abschnitt 1.4. Ich möchte Ihnen zunächst den Sourcecode erklären. Vor dem eigentlichen Programmtext werden die benötigten InterViews-Headerdateien eingebunden. Grundsätzlich gilt folgende Faustregel: Für jede verwendete InterViews-Klasse existiert ein eigenes Headerfile. Der genaue Name dieser Datei, der eventuell von dem Klassennamen leicht abweicht, kann im Referenzmanual [LIN 91] nachgeschlagen werden. Dort findet man zu jeder Klasse am Anfang der Beschreibung die Bezeichnung des entsprechenden Headerfiles.

Damit Sie das Programm besser verstehen können, werden anschließend die verwendeten Klassen einzeln kurz beschrieben. Dazu werden auch die nötigen Teilabschnitte aus den zugehörigen Klassendeklarationen aufgelistet.

Session:

Der an dieser Stelle interessante Deklarationsausschnitt der Klasse Session sieht folgendermaßen aus:

```
class Session
  {
    public:
      Session(const char *classname, int &argc, char **argv,
              const OptionDesc *options = nil,
              const PropertyData *properties = nil);
      Display* default_display() const;
      virtual void run_window(Window*);
      ...
  };
```

Jede InterViews-Anwendung muß als erstes genau ein Session-Objekt erzeugen. Das Session-Objekt übernimmt "Verwaltungsaufgaben" für eine InterViews-Sitzung. So erzeugt dieses Objekt beispielsweise automatisch ein Display-Objekt (wird als nächstes erklärt) für den Standardbildschirm. Das ist in der Regel derjenige, der in der Environmentvariablen DISPLAY enthalten ist. Ein Zeiger auf dieses Display-Objekt wird von der Memberfunktion default_display zurückgeliefert.

Wichtig ist auch die Memberfunktion run_window. Sie bildet das als Zeiger übergebene Window-Objekt (wird ebenfalls später erklärt) auf dem Bildschirm ab. Außerdem wird eine Schleife gestartet, die Eingabeereignisse wie z.B. das Drücken eines Mausknopfs abfrägt.

Die Bedeutung der Parameter des Konstruktors der Klasse Session ist für eine exakte Beschreibung in diesem Tutorial zu umfangreich. Verwenden Sie in Ihren eigenen Programmen als ersten Parameter stets einen String, der irgendwie mit dem Programmnamen zusammenhängt, jedoch nicht unbedingt den Programmnamen selbst. Als zweites und drittes Argument geben Sie argc und argv aus der Kopfzeile der main-Funktion an. Die beiden letzten Parameter sollten Sie einfach weglassen und somit die Voreinstellung nil akzeptieren. Die genaue Bedeutung dieser Parameter ebenso wie die der übrigen Memberfunktionen von Session können Sie als fortgeschrittener InterViews-Anwender einmal selbst im Referenzmanual [LIN 91] nachschlagen.

Display:

Der für uns wichtige Ausschnitt aus der Deklaration der Klasse Display hat folgende Gestalt:

```
class Display
  {
    public:
      virtual Coord width() const;
      virtual Coord height() const;
      virtual Style* style() const;
      ...
  };
```

Ein Objekt dieser Klasse repräsentiert einen Bildschirm. Der Programmierer braucht ein solches Objekt nicht selbst zu erzeugen, da dies bereits von dem Session-Objekt einer InterViews-Applikation erledigt wird. Durch die Session-Memberfunktion default_display erhält er aber Zugriff auf diesen Standardbildschirm. Damit stehen ihm für dieses Objekt die Memberfunktionen der Klasse Display zur Verfügung. Drei davon möchte ich an dieser Stelle erklären.

Die Routinen width und height liefern die Breite und Höhe des zu dem Display-Objekt gehörenden Bildschirms. Die Memberfunktion style gibt einen Zeiger auf das Style-Objekt (wird unten beschrieben) zurück, das automatisch mit dem Display-Objekt erzeugt wird.

Style:

Auch für diese Klasse werden anhand eines Deklarationsausschnitts die anschaulichsten Memberfunktionen kurz erklärt:

```
class Style
  {
    public:
      Style();
      virtual const Color* foreground() const;
      virtual const Color* background() const;
      virtual const Font* font() const;
      ...
  };
```

Styles dienen dazu, gewisse Voreinstellungen zu treffen oder abzufragen, die die Ausgabe der InterViews-Anwendung am Bildschirm betreffen. So liefern die oben angegebenen Memberfunktionen foreground, background und font Zeiger auf Standardfarben für Vorder- und Hintergrundfarbe sowie für einen Standardzeichensatz (repräsentiert durch ein Objekt der Klasse Font). Zu jedem Display-Objekt wird ein Style-Objekt erzeugt, das die Defaulteinstellungen für den entsprechenden Bildschirm enthält. Styles können aber auch unabhängig erzeugt werden. Bestimmte Objekte verlangen nämlich ein Style-Objekt bzw. einen

Zeiger darauf als Parameter. Mit Hilfe des Styles können dem Objekt mit nur einem einzigen Argument zahlreiche Einstellungen wie Vorder- und Hintergrundfarbe, Zeichensatz, etc. mitgegeben werden.

ApplicationWindow:

Jede InterViews-Anwendung öffnet mindestens ein Fenster auf dem Bildschirm. Eines davon muß zum Hauptfenster bestimmt werden. Dieses Fenster wird als Objekt der Klasse ApplicationWindow erzeugt. ApplicationWindow ist von ManagedWindow abgeleitet, das wiederum von der Klasse Window abgeleitet ist (vgl. Abbildung 1.1). Das bedeutet, daß überall, wo ein Zeiger auf ein Window-

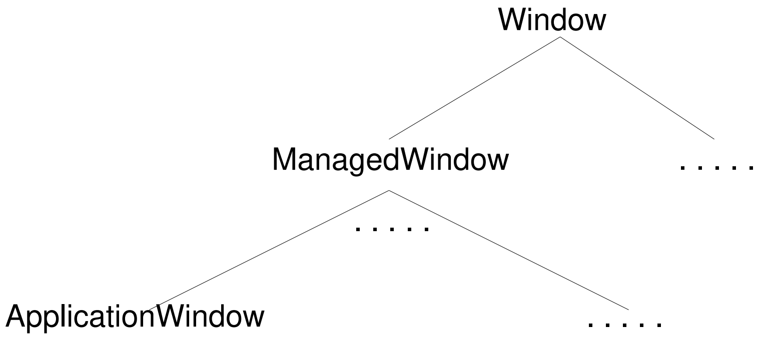

Abbildung 1.1: Ausschnitt aus der Windows-Klassenhierarchie

Objekt als Argument verlangt ist, auch ein Zeiger auf ein ApplicationWindow stehen darf. Insbesondere ist ein Zeiger auf ein ApplicationWindow als Argument von Session::run_window(Window*) erlaubt.

Der interessante Deklarationsausschnitt der Klasse ApplicationWindow ist sehr kurz:

```
class ApplicationWindow : public ManagedWindow
  {
    public:
      ApplicationWindow(Glyph*);
      ...
  };
```

Einem Window bzw. ApplicationWindow wird im Konstruktor ein Zeiger auf ein Glyph-Objekt übergeben. Dieser Glyph wird automatisch in dem Fenster abgebildet. *Glyphs sind die zentralen Objekte in InterViews zum Aufbau einer graphischen Benutzeroberfläche.* Sie werden im folgenden beschrieben.

Glyph:

Die Klasse Glyph ist eine abstrakte Basisklasse, d.h. es können nur Objekte abgeleiteter Klassen direkt erzeugt werden. Fast alle Klassen, die auf dem Bildschirm darstellbare Objekte erzeugen, sind von Glyph direkt oder indirekt abgeleitet. Dies trifft auch auf die im Beispiel verwendeten Klassen Label und Background zu. Abbildung 1.2 zeigt die genaue Ableitungsstruktur. Diese Abbildung zeigt,

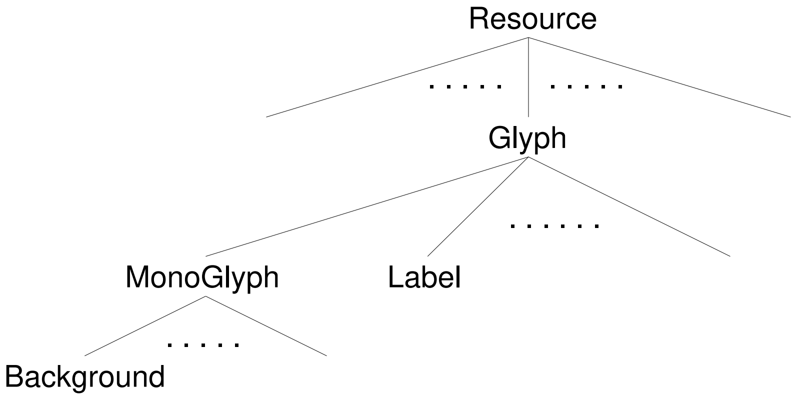

Abbildung 1.2: Einbettung von Glyph, Label und Background in die InterViews-Klassenhierarchie

daß die Klasse Glyph von Resource abgeleitet ist. Damit sind z.B. alle Objekte der Klassen Glyph, Background und Label shared objects.

MonoGlyph:

Wie die Klasse Glyph ist auch diese Klasse abstrakt, wird aber verständnishalber ebenfalls kurz erwähnt. Ein MonoGlyph enthält genau einen Glyph und verändert dessen Aussehen oder Verhalten in einer bestimmten Weise. Ein Objekt der MonoGlyph-Subklasse Background unterlegt beispielsweise einen Glyph mit einem farbigen Hintergrund.

Background:

Diese Klasse ist von MonoGlyph abgeleitet. Sie ändert das Aussehen des enthaltenen Glyph-Objekts, indem unter diesen Glyph ein Hintergrund in einer bestimmten Farbe gezeichnet wird. Wenn beispielsweise einem ApplicationWindow ein Glyph-Objekt übergeben wird, ist es sinnvoll, dieses in ein Background-Objekt zu "packen" und das so entstandene Background-Objekt an das ApplicationWindow zu übergeben. Die meisten Glyphs sind nämlich "durchsichtig", d.h. der Fensterhintergrund ist durch den Glyph sichtbar. Dieser Fensterhintergrund wird aber oft von zufälligen Speichermustern gebildet, was zu unschönen Effekten führen kann.

Sehen wir uns für diese Klasse den interessanten Deklarationsausschnitt an:

```
class Background : public MonoGlyph
  {
    public:
      Background(Glyph*, const Color*);
      ...
  };
```

Die Parameter des Konstruktors sind selbsterklärend. Als erstes Argument wird ein Zeiger auf den Glyph angegeben, unter den der Hintergrund gezeichnet werden soll. Schließlich wird noch ein Zeiger auf das Color-Objekt übergeben, das die Farbe des Hintergrunds bestimmt. Die Klasse Color wird an dieser Stelle noch nicht ausführlicher erklärt, ihre Verwendung wird aber aus dem Beispiel klar.

Label:

Mit Hilfe dieser Klasse werden Objekte erzeugt, die Strings am Bildschirm darstellen. Der wichtigste Teil der Deklaration hat folgende Gestalt:

```
class Label : public Glyph
  {
    public:
      Label(const char*, const Font*, const Color*);
      ...
  };
```

Der Konstruktor erhält als ersten Parameter den darzustellenden String. Die anderen beiden Parameter geben den Zeichensatz und die Farbe an, mit denen das Label abgebildet werden soll.

Beschreibung des Sourcecodes

Mit diesen – doch recht umfangreichen – Informationen sind Sie jetzt in der Lage, das Programm zu verstehen. In der ersten Zeile des Rumpfes von main wird ein Session-Objekt erzeugt. In der zweiten Zeile wird das Style-Objekt ermittelt, das zum Default-Display gehört. In den folgenden Zeilen wird ein ApplicationWindow erzeugt. Durch session–>run_window() wird dieses Fenster auf dem Bildschirm abgebildet und dabei wird eine Ereignisabfrageschleife gestartet. Die genaue Bedeutung dieser Schleife wird in Kapitel 3 beschrieben. Das ApplicationWindow verlangt als Argument einen Zeiger auf ein Glyph-Objekt oder auf ein Objekt einer von Glyph abgeleiteten Klasse. Dieses Objekt wird in dem Fenster dargestellt. Wir verwenden ein Background-Objekt; Background enthält nach Abbildung 1.2 Glyph in seiner Ableitungshierarchie und ist somit als Argument zulässig. Der Background-Konstruktor fordert als Parameter Zeiger auf ein Glyph- und ein Color-Objekt. Als Farbe wählen wir style–>background(), also die Standardhintergrundfarbe des Default-Displays. Als Glyph wird ein Label-Objekt verwendet, das wiederum mit drei Argumenten parametrisiert ist. Das erste ist der String "hello!". Als Zeichensatz und Farbe des Labels werden in diesem Beispiel mit style–>font() und style–>background() ebenfalls zwei Standardeinstellungen gewählt.

Ich muß ehrlich gestehen, daß diese Flut an Informationen für den InterViews-Neuling bestimmt schwer zu verstehen ist. Daher empfehle ich Ihnen, zunächst das Programm wie im folgenden Unterabschnitt beschrieben zum Laufen zu bringen. Lesen Sie danach dieses Kapitel eventuell nochmals durch und beginnen Sie, mit dem Programm zu experimentieren. Vielleicht blättern Sie auch mal im Referenzmanual und sehen sich z.B. an, wie man eine Farbe mittels eines Color-Objekts erzeugt. Ersetzen Sie dann style–>foreground() und style–>background() durch eigene Farben und gewinnen Sie so Sicherheit im Umgang mit InterViews.

1.4 Übersetzung von InterViews-Anwendungen

Wie in C bzw. C++ üblich, werden auch InterViews-Anwendungen in der Regel mit Hilfe eines Makefiles übersetzt. Schwierigkeiten bereiten dabei immer wieder so lästige Dinge wie die Auswahl der richtigen Pfade für Includefiles oder Bibliotheken. Deshalb bietet InterViews die Möglichkeit, Makefiles automatisch zu erzeugen. Grundlage dafür ist ein sogenanntes Imakefile. Grundsätzlich wird für jedes InterViews-Programm ein eigenes Imakefile benötigt. Für unser erstes Beispiel sieht das Imakefile folgendermaßen aus:

```
SPECIAL_IMAKEFLAGS = -DUseInstalled -DTurnOptimizingOn=0

SRCS = example1.$(CCSUFFIX)

OBJS = example1.o

AOUT = sample1

Use_libInterViews()

ComplexProgramTarget(sample1)

MakeObjectRule(example1,example1,NullParameter)
```

Zunächst werden zwei spezielle Makeflags, deren genaue Bedeutung hier nicht interessieren soll, eingestellt. In der folgenden Zeile wird das Makro SRCS mit den Namen der Sourcefiles belegt. Ich habe für das Beispielprogramm aus dem vorhergehenden Abschnitt den Namen example1.c gewählt. Sie können sich selbst einen besseren Dateinamen überlegen. Wählen Sie aber bitte als Endung des Filenamens die Standardextension für C++-Sourcefiles auf Ihrem System. SRCS wird also mit diesem Filenamen besetzt, als Extension wird der Inhalt des Makros CCSUFFIX verwendet, das normalerweise mit der Standardendung für C++-Quelldateien belegt ist.

OBJS wird mit den Namen der Objektfiles besetzt, die bei der Compilierung entstehen. Sinnvollerweise wurde hier example1.o gewählt. Das Makro AOUT erhält den Namen des entstehenden ausführbaren Programms, der in unserem Fall auf "sample1" festgelegt ist. Durch das Makro Use_libInterViews() wird Ihr Programm automatisch mit der InterViews-Standardbibliothek gebunden. Das Makro ComplexProgramTarget erhält als Parameter in der Regel den Namen des zu erzeugenden Programms, in unserem Fall sample1. Für jede Quelldatei muß schließlich noch ein Makro "MakeObjectRule" angegeben werden. Dabei wird dem Compiler mitgeteilt, wie er ein Sourcefile in ein Objektfile umwandeln soll. Da wir diese beiden Dateien in den SRCS- und OBJS-Makros jeweils als example1.$(CCSUFFIX) bzw. example1.o angegeben haben und wir dem C++-Compiler keine weiteren Optionen übergeben wollen, leistet die oben angegebene letzte Zeile des Imakefiles das Gewünschte.

Aus diesem Imakefile wird durch Angabe des Kommandos "ivmkmf" (InterViews make Makefile) ein Makefile erzeugt. Das Kommando "ivmkmf" sollte nach der InterViews-Installation auf Ihrem System zur Verfügung stehen. Falls dies nicht der Fall ist, wenden Sie sich bitte an Ihren Systembetreuer. "ivmkmf" sucht in dem aktuellen Verzeichnis ein Imakefile und übersetzt es in ein Makefile. Durch Eingabe von "make" wird Ihr Programm nun automatisch übersetzt. Da ivmkmf immer nach einem "Imakefile" sucht und ein "Makefile" erzeugt, sollten Sie für die Files eines InterViews-Programms genau ein Verzeichnis öffnen, um so Namenskonflikte bei den Imakefiles und Makefiles verschiedener Programme zu vermeiden.

Geben Sie jetzt bitte das InterViews-Beispiel aus dem vorhergehenden Unterabschnitt und das obige Imakefile in Ihren Rechner ein. Erzeugen sie mittels "ivmkmf" ein Makefile und starten Sie anschließend "make". Wenn Sie nach dem erfolgreichen Übersetzen Ihr Verzeichnis ansehen, werden Sie das ausführbare File "sample1" entdecken. Starten Sie das Programm und freuen Sie sich über Ihre erste InterViews-Applikation. Sie können das Programm nur durch Zerstören des Prozesses von außen beenden, indem Sie z.B. <ctrl>–c in dem Fenster drücken, in dem Sie sample1 gestartet haben. Dies liegt daran, daß wir durch session–>run_window(...) eine Eingabeabfrageschleife gestartet haben, die in unserem Beispiel nirgendwo verlassen werden kann. Wie ein InterViews-Programm regulär verlassen wird, werden Sie später lernen.

Kapitel 2

Ausgabe auf den Bildschirm

In diesem Kapitel lernen Sie, Graphiken auf dem Bildschirm darzustellen, die aus einfachen Bestandteilen wie z.B. Linien oder Polygonen bestehen. Dazu werden wir uns zunächst eine kleine Beispielaufgabe definieren. Nach der Diskussion der nötigen InterViews-Klassen wird die Implementation dieser Aufgabenstellung besprochen. Zum Abschluß des Kapitels möchte ich noch einige Anregungen zum Experimentieren geben.

2.1 Aufgabenstellung

Mir kommt es in dem zweiten Kapitel dieses Tutorials nur darauf an, Ihnen prinzipiell zu erklären, wie in InterViews Bildschirmausgaben erzeugt werden. Deshalb wollen wir hier exemplarisch nur eine relativ einfache Aufgabe lösen. Auf dem Bildschirm soll ein Fenster erscheinen, das eine schwarze Linie von links unten nach rechts oben enthält. Zusätzlich soll dieses Fenster vom Benutzer in bestimmten Schranken vergrößert und verkleinert werden können. Unabhängig von der Fenstergröße soll dabei die Linie aber stets automatisch vom linken unteren zum rechten oberen Fensterrand gezeichnet werden. Abbildung 2.1 zeigt einen Screendump der Lösung. Das Aussehen des Fensterrahmens ist dabei abhängig von dem verwendeten Windowmanager und kann demzufolge auf Ihrem System anders gestaltet sein. Wichtig ist der Inhalt des Fensters, der von unserem InterViews-Programm erzeugt werden soll.

2.2 Notwendige InterViews-Klassen

Zunächst ist wieder etwas Theorie nötig. Ich empfehle Ihnen, vor dem Studium dieses Abschnitts das Programm, das die gegebene Aufgabenstellung löst, einzugeben und ablaufen zu lassen. Wenn Sie mit dem Programm experimentieren, indem Sie z.B. das abgebildete Fenster vergrößern oder verkleinern, werden Sie

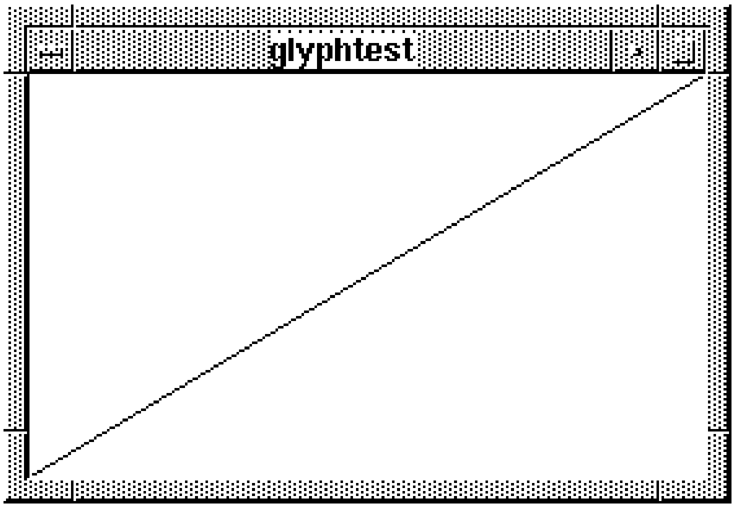

Abbildung 2.1: Bildschirmausgabe des Beispielprogramms von Kapitel 2

diesen theoretischen Abschnitt besser verstehen können. Den Programmtext und das zugehörige Imakefile finden Sie im Abschnitt 2.3.

Im folgenden werden die InterViews-Klassen vorgestellt, die zur Lösung der Aufgabe nötig sind. Die grundlegenden Klassen wie Session oder Window, die in jeder InterViews-Applikation verwendet werden, wurden bereits im ersten Kapitel soweit erforderlich beschrieben. Daher werden hier nur die Klassen angegeben, die speziell für die Bildschirmausgabe gebraucht werden.

Dennoch werden manche Punkte, die Sie schon früher kennengelernt haben, nochmals wiederholt. Solche Wiederholungen werden Sie an mehreren Stellen über das ganze Buch hinweg bemerken. Ich möchte damit erreichen, daß Sie sich wichtige Sachverhalte besser einprägen können.

Color:

Objekte der Klasse Color dienen der Erzeugung von Farben. Der für uns wichtige Deklarationsausschnitt hat folgende Gestalt:

```
typedef unsigned int ColorOp;
typedef float ColorIntensity;

class Color : virtual public Resource
  {
        public:
          enum{Copy, Xor};
          Color(ColorIntensity red, ColorIntensity green,
                ColorIntensity blue, float alpha = 1.0,
```

```
                ColorOp = Copy);
        ...
};
```

An dieser Stelle möchte ich kurz eine Bemerkung zur Notation machen: Wie Sie wissen, ist bei der Deklaration von Klassen und Funktionen die Angabe von Parameterbezeichnern hinter den Typangaben überflüssig. Aus diesem Grund werden Sie in den Deklarationsabschnitten meist weggelassen. Ist aber der angegebene Parametertyp nicht aussagekräftig genug, wird häufig noch eine möglichst anschauliche Parameterbezeichnung hinzugefügt.

Damit wollen wir uns aber wieder der Klasse Color zuwenden. Color ist von Resource abgeleitet, d.h. *Objekte dieser Klasse gehören zu den shared objects und dürfen nicht mit delete gelöscht werden!*

Von den Memberfunktionen ist im Moment nur der Konstruktor wichtig, die übrigen können Sie selbst im Referenzmanual [LIN 91] nachschlagen. Die ersten drei Argument geben den Rot-, Grün- und Blauanteil der zu erzeugenden Farbe an. Sie können jeweils beliebige Floatwerte zwischen 0.0 und 1.0 angeben. 1.0 steht für die volle Intensität des entsprechenden Farbanteils. 0.0 zeigt dagegen an, daß der betreffende Farbanteil in der erzeugten Farbe gleich Null ist.

Die beiden letzten Argumente werden normalerweise nicht angegeben, weil die eingestellten Defaultwerte den Effekt erzeugen, den man intuitiv erwartet. Mit dem vierten Parameter könnten Sie die "Sichtbarkeit" der Farbe regeln. Dabei sind wieder Werte zwischen 0.0 und 1.0 möglich, z.B. bedeutet alpha = 1.0, daß die Farbe "voll sichtbar" ist. Der Wert Copy für den letzten Parameter bewirkt, daß beispielsweise mit Linien einer so erzeugten Farbe der Hintergrund übermalt wird. Wenn Sie über einen Monochrombildschirm verfügen und Xor angeben, wird die Linie mit dem Hintergrund exklusiv-oder-verknüpft. Dies bedeutet, daß bereits dunkle Bildschirmpunkte wieder hell werden, wenn Sie darüber eine dunkle Linie zeichnen.

Wenn Sie eine bestimmte Farbe erzeugen wollen, müssen Sie bei der Angabe der Rot-, Grün- und Blauanteile darauf achten, daß wir es hier mit additiver Farbmischung zu tun haben. Zur Erleichterung sind einige Argumenttripel mit den zugehörigen Farben angegeben:

```
new Color(0.0,0.0,0.0);  \\ Schwarz;
new Color(1.0,1.0,1.0);  \\ Weiss;
new Color(1.0,0.0,0.0);  \\ helles Rot;
new Color(0.0,0.0,0.5);  \\ dunkles Blau;
new Color(0.8,0.0,0.8);  \\ Lila.
```

Eine sehr schöne Farbtafel, die die Zusammensetzung vieler Mischfarben übersichtlich darstellt, ist in [PRO 87] auf Seite 76 zu finden.

Bitte beachten Sie, daß aufgrund eines InterViews-Fehlers innerhalb einer Applikation nicht mehr als ca. 250 Farben korrekt erzeugt werden können. Die

Erzeugung weiterer Farben ist zwar möglich, doch entsprechen diese nicht mehr der Farbspezifikation, die Sie im Konstruktor angegeben haben.

Brush:

Objekte der Klasse Brush werden vielen Zeichenoperationen übergeben, um Strichstärke und Strichmuster zu bestimmen. Somit entspricht ein Brush der Wirkung eines "Pinsels". Anhand eines Deklarationsausschnitts soll diese Klasse kurz erläutert werden:

```
class Brush : virtual public Resource
  {
    public:
      Brush(Coord width);
      ...
  };
```

Wie Color ist auch Brush von Resource abgeleitet. Demzufolge gehören Objekte der Klasse Brush ebenfalls zu den shared objects.

Für unsere Zwecke reicht der oben aufgeführte Konstruktor aus. Als Parameter wird ihm die gewünschte Strichstärke des zu erzeugenden Pinsels übergeben. Es gibt noch zwei weitere Konstruktoren, mit deren Hilfe auch noch das Strichmuster spezifiziert werden kann. Sehen Sie bitte bei Interesse im Referenzmanual [LIN 91] nach.

Canvas:

Ein Canvas-Objekt repräsentiert eine "Leinwand", auf die gezeichnet werden kann. Während Glyphs spezifizieren, *was gezeichnet wird*, stellt ein Canvas dagegen die Zeichenfläche zur Verfügung, d.h. er steht für das Objekt, *auf das gezeichnet wird*. Die von InterViews bereitgestellten Fensterobjekte, z.B. ApplicationWindows, erzeugen automatisch ein Canvas-Objekt. Die Leinwand wird dann von Glyphs "bemalt". Dieser Mechanismus wird auch zur Lösung unserer Aufgabe verwendet, wie Sie in Abschnitt 2.3 sehen werden. Der vorgestellte Deklarationsausschnitt fällt für die Klasse Canvas etwas umfangreicher aus:

```
class Canvas
  {
    public:
      virtual Window* window();
      virtual void new_path();
      virtual void move_to(Coord x, Coord y);
      virtual void line_to(Coord x, Coord y);
      virtual void close_path();
```

```
    virtual void stroke(const Color*, const Brush*);
    virtual void fill(const Color*);
    ...
};
```

Der Konstruktor wurde hier absichtlich nicht angegeben, obwohl er ebenfalls als öffentliche Memberfunktion verfügbar ist. Da die Objekte der Klassen, die InterViews für Bildschirmfenster bereitstellt (Window und davon abgeleitete Klassen), jeweils selbst einen Canvas erzeugen (siehe oben), braucht der InterViews-Programmierer dies im Regelfall nicht selbst zu tun. Die Memberfunktion window liefert einen Zeiger auf dieses Fenster, zu dem das Canvas-Objekt gehört.

Die übrigen, oben angeführten Memberroutinen dienen dazu, direkt auf den Canvas zu zeichnen. Die Zeichenoperationen sind allerdings etwas umständlich. Canvas stellt zwar noch wenige weitere, mächtigere Funktionen zur Verfügung, am bequemsten läßt sich jedoch mit der in Abschnitt 2.4 vorgestellten Klasse Painter arbeiten.

Die Klasse Painter gehört aber zu den sogenannten "2.6–kompatiblen Klassen". Diese Bezeichnung bezieht sich auf die Versionsnummer der Vorgängerversion von InterViews 3.0. Bei der Umstellung von Version 2.6 auf 3.0 wurden größere Veränderungen vorgenommen. Leider sind viele Klassen noch nicht auf die neue Version umgestellt worden. Sie können zwar durch einfache Maßnahmen, die in Abschnitt 2.4 beschrieben werden, noch in 3.0-Applikationen verwendet werden, doch halte ich eine Mischung aus Konzepten zweier verschiedener Versionen für unschön. Die InterViews-Autoren haben bisher aber noch keine verbindlichen Programmierrichtlinien dafür angegeben, ob die Mischung von Klassen beider Versionen erlaubt sein soll oder nicht. Daher schlage ich an dieser Stelle einen Mittelweg ein. In Abschnitt 2.4 wird der bequemere, aber unsaubere Weg mittels der 2.6–kompatiblen Klasse Painter gewählt. Im folgenden wird der umständlichere, jedoch voll Version 3.0–kompatible Weg beschritten. Sie können sich die Variante aussuchen, die Ihnen besser gefällt. Außerdem bleibt zu hoffen, daß in der nächsten InterViews-Version 3.1 alle Klassen einheitlich als "3.1–kompatible" Klassen zur Verfügung stehen oder verbindliche Richtlinien für die Verwendung alter Klassen herausgegeben werden.

Jetzt wollen wir uns also ansehen, wie mit den einfachen Memberroutinen der Klasse Canvas Zeichnungen erzeugt werden können:

- Zunächst wird mit new_path() ein sogenannter "Pfad" geöffnet.
- Danach wird mit move_to() der Ausgangspunkt des Pfades festgelegt.
- Durch nachfolgende line_to()-Aufrufe kann ein beliebiger Pfadverlauf spezifiziert werden.
- Mittels close_path() wird der Pfad abgeschlossen.

- Der so angegebene Pfad kann auf zwei Arten am Bildschirm abgebildet werden:
  - Durch stroke(...) wird er mit der übergebenen Farbe und dem übergebenen Pinsel als Linienzug gezeichnet.
  - Der Aufruf von fill(...) bewirkt, daß die von dem Pfad eingeschlossene Fläche mit der angegebenen Farbe ausgefüllt wird.

  Um beispielsweise eine schwarz umrandete, weiß gefüllte Fläche zu erzeugen, kann die Kombination beider Aufrufe sinnvoll sein.

Als Beispiel sehen wir uns folgende Pfaddefinition an:

```
canvas->new_path();      // canvas ist vom Typ Canvas*
canvas->move_to(x1,y1);  // x1, y1 vom Typ Coord
canvas->line_to(x2,y2);  // x2, y2 vom Typ Coord
canvas->line_to(x3,y3);  // x3, y3 vom Typ Coord
canvas->close_path();

// canvas->stroke(...) oder canvas->fill(...)
```

Im Anschluß an diese Pfaddefinition können Sie stroke oder fill aufrufen. Das Ergebnis dieser beiden Routinen ist in Abbildung 2.2 zu sehen.

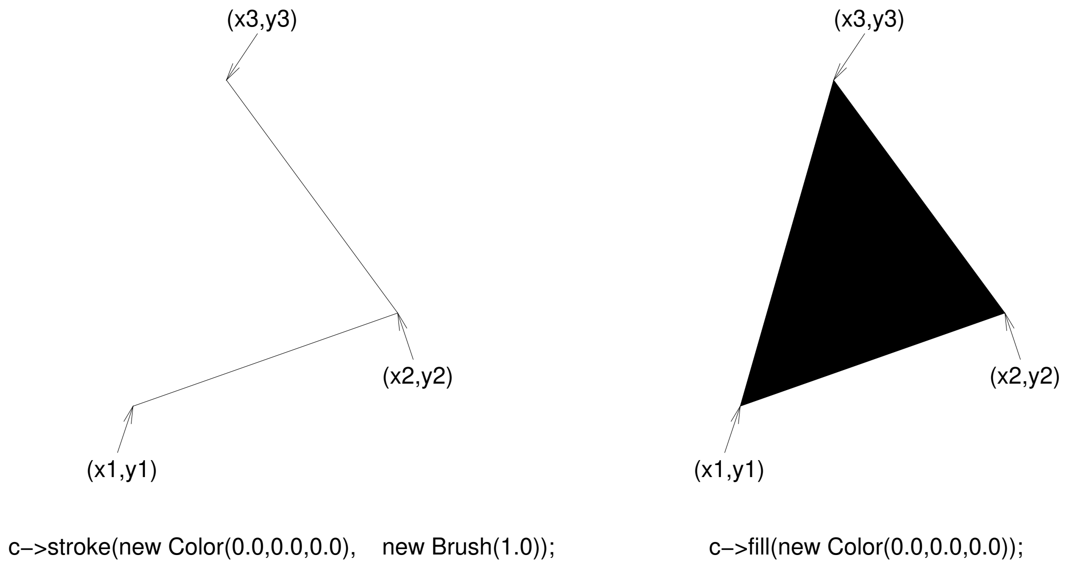

Abbildung 2.2: Auswirkung der Routinen stroke und fill auf einen zuvor spezifizierten Pfad

Glyph:

Objekte dieser Klasse sind die elementaren Bausteine für graphische Oberflächen unter InterViews, wie bereits in Kapitel 1 angedeutet wurde. Ebenfalls in Kapitel 1 wurde erklärt, daß einem Window-Objekt ein Zeiger auf einen Glyph übergeben wird. Wird das Fenster auf dem Bildschirm abgebildet, so wird automatisch dieser Glyph gezeichnet.

An dieser Stelle möchte ich eine Konvention einführen: Wenn im weiteren von "Glyphs" gesprochen wird, so sind damit Objekte der Klasse Glyph oder Objekte von Klassen gemeint, die von Glyph abgeleitet sind. Außerdem wird in den Fällen, in denen Zeiger auf Glyph-Objekte als Parameter angegeben werden müssen, häufig einfach von der "Übergabe eines Glyphs" gesprochen.

Glyphs verdienen aufgrund ihrer besonderen Bedeutung eine ausführliche Behandlung, daher enthält dieses Buch in Kapitel 5 einen eigenen Abschnitt über dieses Thema.

Wir wollen uns nun anhand eines Deklarationsausschnitts die elementaren Memberfunktionen der Klasse ansehen:

```
class Glyph : virtual public Resource
  {
    protected:
      Glyph();

    public:
      virtual void request(Requisition&) const;
      virtual void allocate(Canvas*,const Allocation&,
                            Extension&);
      virtual void draw(Canvas*,const Allocation&) const;
      ...
  };
```

Der Konstruktor der Klasse ist geschützt, d.h. er steht nur abgeleiteten Klassen zur Verfügung. Damit können Sie selbst kein Objekt der Klasse Glyph erzeugen. Dies ist auch sinnvoll, da diese Basisklasse nur ein "Protokoll" für Objekte anbieten soll, die in einem Bildschirmfenster abgebildet werden. Die oben angegebenen Memberfunktionen regeln die Kommunikation zwischen "Glyphs" und dem System. Das konkrete Verhalten dieser virtuellen Funktionen müssen Sie selbst implementieren. Dies geschieht dadurch, daß Sie von Glyph eigene Klassen ableiten und die virtuellen Funktionen neu implementieren. *Die grundsätzliche Vorgehensweise von InterViews-Programmierern besteht darin, von solchen "Gerüstklassen" eigene Klassen abzuleiten. Durch Implementation bzw. Reimplementation der virtuellen Funktionen und Hinzufügen neuer Memberfunktionen können sie die abgeleiteten Klassen ihren Vorstellungen entsprechend gestalten.*

InterViews bietet zu Ihrer Arbeitserleichterung bereits viele solcher abgeleiteter Klassen an, z.B. die von Glyph abgeleitete Klasse Label, die Sie bereits im Beispiel aus Kapitel 1 kennengelernt haben.

Im folgenden wird nun das Glyph-Protokoll anhand der drei wichtigsten Memberfunktionen request, allocate und draw besprochen. An dieser Stelle wird aber nur ein grober Überblick gegeben. Eine ausführlichere Erklärung folgt im Abschnitt 2.3.

Wenn ein Fenster auf dem Bildschirm abgebildet werden soll, versucht das System, den Glyph zu zeichnen, der dem Window-Objekt im Konstruktor übergeben wurde. Dabei laufen im einzelnen folgende Schritte ab:

- Zunächst ruft das Laufzeitsystem die Memberfunktion request auf. In request muß der Glyph in dem übergebenen Parameter "Requisition&" den Koordinatenbereich angeben, den er auf dem Bildschirm einnehmen will. Mit diesen Daten wird dann beispielsweise automatisch die Größe des Bildschirmfensters berechnet.

- Danach ruft das System allocate auf. In dieser Funktion wird dem Glyph der Canvas mitgeteilt, auf den er zeichnen soll. In dem Parameter "Allocation&" wurde vom System der Koordinatenbereich abgespeichert, der dem Glyph aufgrund seiner Anforderung in request tatsächlich zugeteilt wurde. Dieser Bereich kann von der Anforderung abweichen. Der Parameter "Extension&" wird später erklärt.

- Zum Abschluß wird draw mit denselben Parametern "Canvas*" und "const Allocation&" wie zuvor allocate aufgerufen. In dieser Routine sollte der Glyph die eigentlichen Zeichenoperationen durchführen.

Der Programmierer ruft im Regelfall keine dieser drei Memberfunktionen selbst auf. So hat beispielsweise ein expliziter Aufruf von draw im Regelfall keinen Effekt.

Nun fehlt noch die Beschreibung der Klassen Requisition, Allocation und Extension. Diese folgt in Abschnitt 2.3.

2.3 Implementierung der Aufgabenstellung

Die eigentliche Implementierung unserer Aufgabe ist wesentlich weniger umfangreich, als man es nach dieser doch recht ausführlichen Klassenbeschreibung vermuten könnte. Der Grund dafür ist, daß InterViews wirklich sehr einfach zu programmieren ist, "wenn man weiß, wie es funktioniert". Und um das zu lernen, läßt sich ein gewisser Aufwand leider nicht vermeiden. Aber Sie werden sehen, daß es sich lohnt!

Die Lösung unserer Aufgabe ist in die vier Files header.h, Linie.h, Linie.c und glyphtest.c unterteilt. Die Datei header.h enthält eine Zusammenfassung der benötigten Headerfiles:

```
#ifndef header_h
#define header_h

#include <InterViews/session.h>
#include <InterViews/display.h>
#include <InterViews/style.h>
#include <InterViews/window.h>
#include <InterViews/glyph.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <iostream.h>

#include "Linie.h"

#endif
```

Die Datei "iostream.h" wurde hier aufgenommen, weil ich in der Testphase des Programms verschiedene cerr-Meldungen in den Programmtext eingefügt hatte.

Im übrigen ist die Handhabung der Headerfiles Geschmackssache. Mir ist es am liebsten, alle innerhalb eines Programms verwendeten Headerdateien in einer eigenen Headerdatei zusammenzufassen. Sie könnten aber auch die jeweils benötigten Header einzeln in Linie.c oder glyphtest.c angeben. Mit meiner Vorgehensweise genügt es jedoch, dort lediglich "header.h" zu inkludieren.

Die Datei Linie.h enthält die Deklaration der von Glyph abgeleiteten Klasse Linie:

```
#ifndef Linie_h
#define Linie_h

class Linie : public Glyph
{
  public:
    Linie();
    virtual void request(Requisition&) const;
    virtual void allocate(Canvas*,const Allocation&,
                              Extension&);
    virtual void draw(Canvas*,const Allocation&) const;
};

#endif
```

Diese Klasse wird dafür verantwortlich sein, die gewünschte Linie von links unten nach rechts oben zu zeichnen. Dazu werden die drei virtuellen Glyph-Memberfunktionen request, allocate und draw neu definiert.

Das eigentliche Hauptprogramm ist in dem File glyphtest.c enthalten:

```
#include "header.h"

int main(int argc,char **argv)
{
  Session *session = new Session("GlyphExample",argc,argv);
  Style *style = session->style();

  Linie *linie = new Linie();

  Window *window = new ApplicationWindow(linie);

  return session->run_window(window);
}
```

Die Struktur von glyphtest.c ist derjenigen des Beispiels aus Kapitel 1 sehr ähnlich. Die Zeile

```
  Style *style = session->style();
```

ist hier eigentlich überflüssig, weil nirgends auf style zugegriffen wird. Für spätere Erweiterungen (vgl. Abschnitt 2.4) ist diese Zeile aber nützlich und wurde daher an dieser Stelle aufgenommen. Die Session-Memberfunktion style wurde übrigens bisher noch nicht erwähnt. Sie liefert einen Zeiger auf das Style-Objekt zurück, das zu dem Defaultdisplay gehört. Somit ist die Wirkung der obigen Zeile mit dem Effekt von

```
  Style *style = session->default_display->style();
```

identisch.

In glyphtest.c wird wie in dem Beispielprogramm aus Kapitel 1 (example1.c) ein ApplicationWindow erzeugt und mittels session–>run_window auf den Bildschirm gebracht und eine Eingabeabfrageschleife gestartet. Im Unterschied zu example1.c wird dem Fenster hier aber kein Zeiger auf ein Label-Objekt, sondern ein Zeiger auf ein Objekt unserer eigenen Klasse Linie übergeben. Da Linie von Glyph abgeleitet ist, ist dieses Argument für den Konstruktor von ApplicationWindow zugelassen. Außerdem ist hier der Glyph (unser Linie-Objekt) im Unterschied zu Kapitel 1 nicht von einem Background-MonoGlyph umgeben. Grund dafür ist, daß das Linie-Objekt seinen Hintergrund selbst zeichnet (siehe unten). Die Methode, den Hintergrund von einem Background-Objekt zeichnen zu lassen, ist jedoch eleganter und wird im Regelfall auch verwendet.

Das File Linie.c leistet die eigentliche Lösung der Aufgabenstellung. Es enthält die Implementierung der Klasse Linie:

```
#include "header.h"

Linie::Linie() : Glyph()
{}

void Linie::request(Requisition &r) const
{
  Requirement rx(250.0,150.0,100.0,0.0);
  Requirement ry(150.0,50.0,0.0,0.0);

  r.require(Dimension_X, rx);
  r.require(Dimension_Y, ry);
}


void Linie::allocate(Canvas*,const Allocation &a,
                             Extension &ext)
{
  ext.xy_extents(a);
}


void Linie::draw(Canvas *c,const Allocation &a) const
{
  Color *black = new Color(0.0,0.0,0.0);
  Color *white = new Color(1.0,1.0,1.0);
  Brush *brush = new Brush(1.0);

  Coord x1 = a.left();
  Coord y1 = a.bottom();
  Coord x2 = a.right();
  Coord y2 = a.top();

  c->new_path();       // Hintergrund zeichnen
  c->move_to(x1,y1);
  c->line_to(x2,y1);
  c->line_to(x2,y2);
  c->line_to(x1,y2);
  c->close_path();
```

```
  c->fill(white);

  c->new_path();        // Linie zeichnen
  c->move_to(x1,y1);
  c->line_to(x2,y2);
  c->close_path();

  c->stroke(black,brush);
}
```

Der Konstruktor ruft lediglich den Konstruktor der Basisklasse Glyph auf. Interessanter ist die Memberfunktion request, in der ein Objekt der Klasse Linie den für sich benötigten Platz anfordert. Diese Anforderung wird in das Requisition-Objekt geschrieben, das der Funktion als Referenzparameter übergeben wurde.

Die Klasse Requisition soll anhand eines kurzen Deklarationsabschnitts vorgestellt werden:

```
typedef enum DimensionName {Dimension_X, Dimension_Y,
                            Dimension_Z};

class Requisition
  {
    public:
      void require(DimensionName, const Requirement&);
      ...
  };
```

Ein Requisition-Objekt umfaßt eine Anforderung für jede Dimension. Diese Teilanforderungen werden in Objekten der Klasse Requirement festgelegt. Somit enthält ein Requisition-Objekt für jede Dimension genau ein Requirement-Objekt. Da in unserem Beispiel ein Glyph auf eine zweidimensionale Leinwand (Canvas) zeichnen soll, werden wir je ein Requirement für die x- und y-Richtung definieren. Die Requirements können mit der Memberfunktion require unter Angabe des zugehörigen Dimensionsnamens festgelegt werden.

Der wichtige Teil der Deklaration der Klasse Requirement sieht wie folgt aus:

```
class Requirement
  {
    public:
      Requirement(Coord natural, Coord stretch,
                  Coord shrink, float align);
      ...
  };
```

Für Objekte dieser Klasse sind vier verschiedene Konstruktoren und etliche Memberfunktionen verfügbar. Normalerweise genügt der hier angegebene Konstruktor, dessen Parameter nun im einzelnen erklärt werden.

Der erste Parameter gibt die "natürliche Größe" eines Objektes an. Aus verschiedenen Gründen kann aber InterViews einem Objekt häufig nicht seine natürliche Größe geben. Stellen Sie sich z.B. vor, daß ein Glyph in einem Fenster auf dem Bildschirm abgebildet wird. Dieser Glyph werde anfangs in seiner natürlichen Größe dargestellt. Der Windowmanager gestattet Ihnen nun, das Fenster in gewissen Schranken zu verkleinern oder zu vergrößern. Sie erwarten selbstverständlich, daß auch der Glyph entsprechend verkleinert bzw. vergrößert wird. Genau diese Schranken werden in den beiden Konstruktorparametern stretch (engl. stretchability = Dehnbarkeit) und shrink (engl. shrinkability = Stauchbarkeit) festgelegt. Mit dem Parameter align können Sie für jede Dimension die Lage des Ursprungs des Glyph-Objekts festlegen. Für align sind Werte zwischen 0.0 und 1.0 erlaubt. 0.0 heißt, daß der Ursprung in der betreffenden Dimension bei der kleinsten Koordinate des Objekts liegt. Entsprechend setzt der Wert 1.0 den Ursprung bei der größten Koordinate fest. *In InterViews wachsen Koordinaten in x-Richtung von links nach rechts und in y-Richtung von unten nach oben.*

Im folgenden sind einige Paare für align-Werte (ax,ay) mit der dazugehörigen Ursprungsdefinition angegeben. ax gibt den align-Wert des Requirement-Objekts in x-Richtung an. Entsprechend enthält ay den align-Wert für die y-Richtung:

```
(0.0,0.0)           links unten
(0.0,0.5)           links mitte
(1.0,0.0)           rechts unten
(1.0,1.0)           rechts oben
```

Wenn Sie das jetzt noch nicht genau verstanden haben, darf ich Sie auf Kapitel 5 vertrösten. Dort wird dieser Sachverhalt nochmals erklärt. In unserem Beispiel spielt der Ursprung des Linie-Objekts keine Rolle. Wir legen ihn daher willkürlich durch die Angabe von align = 0.0 für beide Dimensionen in die linke untere Ecke. Wichtig wird der Ursprung erst, wenn einzelne Glyphs zu komplexen Glyphs (z.B. Boxen) zusammengebaut werden. Dann beeinflußt die Lage der Ursprünge der einzelnen Glyphs das Aussehen des zusammengesetzten Glyphs. Auch das wird ausführlich in Kapitel 5 beschrieben.

Mit diesen Informationen ist jetzt die Memberfunktion Linie::request leicht zu verstehen. Zunächst wird je ein Requirement-Objekt für jede Dimension erzeugt. Die natürliche Größe wird in x-Richtung auf 250.0 (in Coords) festgelegt. Das Objekt soll in x-Richtung um 150 Koordinateneinheiten dehnbar und um 100 Koordinateneinheiten stauchbar sein, d.h. es ist maximal 400 und minimal 150 Koordinateneinheiten breit. Entsprechendes gilt für die y-Richtung. Mit r.require(...) werden die beiden Requirement-Objekte rx und ry in das Requisition-Objekt r eingebaut.

Obwohl der Rumpf von Linie::allocate nur eine Zeile umfaßt, wird die Erklärung dieser Memberfunktion ebenfalls etwas breiteren Raum einnehmen, weil ich an dieser Stelle die beiden Klassen Allocation und Extension erklären möchte.

Ebenso wie Linie::request wird Linie::allocate vom System aufgerufen (vgl. Abschnitt 2.2). Die Memberfunktion allocate wird nicht nur dann aufgerufen, wenn das Linie-Objekt zum erstenmal auf dem Bildschirm erscheint, sondern jedesmal, wenn sich etwas ändert, z.B. wenn das zugehörige Fenster vergrößert wird. Dabei wird dem Linie-Objekt in dem Referenzparameter "const Allocation &a" der Koordinatenbereich übermittelt, der ihm tatsächlich zugeteilt wurde. Von der Deklaration der Klasse Allocation ist folgender Ausschnitt interessant:

```
class Allocation
  {
    public:
      Allocation();
      void allot(DimensionName, const Allotment&);
      Allotment& allotment(DimensionName);
      Coord left() const;
      Coord right() const;
      Coord bottom() const;
      Coord top() const;
      ...
  };
```

Ebenso wie ein Requisition-Objekt enthält auch ein Allocation-Objekt für jede Dimension je ein spezielles Objekt. Für die Klasse Requisition waren diese Objekte Instantiierungen der Klasse Requirement. Dieselbe Rolle spielt bei den Allocations die Klasse Allotment. Sie enthält die Allokierungsinformation für eine bestimmte Dimension. Die Allotment-Objekte für die jeweiligen Dimensionen werden unter Angabe des Dimensionsnamens mittels Allocation::allot bzw. Allocation::allotment gesetzt bzw. zurückgegeben.

Im folgenden wird die Klasse Allotment diskutiert:

```
class Allotment
  {
    public:
      Coord begin() const;
      Coord end() const;
      Coord span() const;
      float alignment() const;
      Coord origin() const;
      ...
  };
```

Die Memberfunktionen erlauben den Zugriff auf die charakteristischen Größen eines Allotment-Objekts. begin liefert die kleinste Koordinate des Koordinatenbereichs, der durch das Allotment-Objekt repräsentiert wird. Analog liefern end bzw. span dessen größte Koordinate bzw. Breite. origin liefert die absolute und alignment die relative Ursprungsposition. Für ein Allotment-Objekt a könnten beispielsweise folgende Werte gelten:

```
a.span() = 100.0;
a.origin() = 20.0;
a.alignment() = 0.4;
a.begin() = -20.0;
a.end() = 80.0.
```

Die Memberfunktionen der Klasse Allocation, die bisher noch nicht erklärt wurden, erleichtern die lesenden Zugriffe auf die in den einzelnen Allotments gespeicherten Daten. Die Namen der Routinen sind selbsterklärend, daher wird hier nur ein kleines Beispiel zum Verständnis vorgeführt:

Sei z.B. alloc ein Allocation-Objekt. Dann ist alloc.left() gleichbedeutend mit alloc.allotment(Dimension_X).begin().

Für das Verständnis der Parameterleiste der Funktion Linie::allocate fehlt nur noch eine Erklärung des Parameters vom Typ "Extension&". Dieser soll jetzt kurz beschrieben werden. In einer älteren Version des InterViews Referenzmanuals [LIN 91] wird erklärt, daß ein Glyph in diesem Parameter seinen "tatsächlich benötigten Platz" festlegen soll. Diese Angabe ist aber unsinnig, da bereits in der Memberfunktion request der Platzbedarf des Glyphs bekanntgegeben wird. Nach Aussage der Autoren des Manuals ist die Existenz dieses Parameters aufgrund der InterViews-Implementierung erforderlich und soll später einmal wegfallen. Daher möchte ich an dieser Stelle die Deklaration der Klasse Extension nicht aufschreiben, sondern nur die einzig nötige Memberfunktion Extension::xy_extents erklären. Sie erhält als Argument ein Allocation-Objekt und setzt das Extension-Objekt auf dieselben Werte. Damit geben Sie quasi an, daß der Glyph den Platz, der ihm vom System zugewiesen wurde, tatsächlich benötigt. Die Zeile ext.xy_extents(a) in Linie::allocate erfüllt diese Aufgabe. Diese Zeile sollte in jeder Ihrer selbstgeschriebenen allocate-Routinen ganz am Anfang stehen.

Nun fehlt nur noch die Beschreibung von Linie::draw. Diese Routine bewirkt die eigentliche Ausgabe auf dem Bildschirm. Die beiden Memberfunktionen Linie::request und Linie::allocate waren dagegen für die Erledigung von "Verwaltungsaufgaben" zuständig.

Zur Erinnerung: Linie::draw wird vom System im Anschluß an Linie::allocate mit denselben Canvas*- und const Allocation&-Parametern aufgerufen. Den Rumpf von draw sollten Sie jetzt schon ohne meine Hilfe verstehen können. Trotzdem möchte ich nochmals kurz den Sourcecode erklären.

In den ersten drei Zeilen werden die Farben Schwarz und Weiß sowie ein Pinsel der Breite 1.0 erzeugt. Anschließend werden die Koordinatenpaare x1 und y1 bzw. x2 und y2 mit den Koordinaten der linken unteren bzw. rechten oberen Ecke des vom System übergebenen Allocation-Objekts besetzt. Die nächsten sieben Zeilen füllen den Hintergrund mit der Farbe Weiß aus. Dazu wird zunächst ein Pfad von der linken unteren über die rechte untere und rechte obere zur linken oberen Ecke spezifiziert. Die von diesem Pfad umschlossene Fläche wird durch den fill-Aufruf weiß gefüllt. In den abschließenden fünf Zeilen wird schließlich noch die schwarze Linie von der linken unteren in die rechte obere Ecke gezeichnet.

Um das angegebene Programm zu kompilieren, kopieren Sie sich das Imakefile aus Abschnitt 1.3 in das Verzeichnis, in dem header.h, Linie.h, Linie.c und glyphtest.c stehen. Ändern Sie das Imakefile wie folgt:

```
SPECIAL_IMAKEFLAGS = -DUseInstalled -DTurnOptimizingOn=0

SRCS = \
    Linie.$(CCSUFFIX) \
    glyphtest.$(CCSUFFIX)

OBJS = \
    Linie.o \
    glyphtest.o

AOUT = glyphtest

Use_libInterViews()

ComplexProgramTarget(glyphtest)

MakeObjectRule(Linie,Linie,NullParameter)
MakeObjectRule(glyphtest,glyphtest,NullParameter)
```

Erzeugen Sie aus diesem Imakefile durch "ivmkmf" ein Makefile. Versuchen Sie nun, das Programm zu übersetzen und auszuführen.

2.4 Anregungen

Zum Abschluß dieses Kapitels will ich Ihnen noch einige Anregungen zum Experimentieren geben.

Bereits in Abschnitt 2.3 wurde schon einmal erwähnt, daß die in Linie::draw gezeigte Methode, den Hintergrund eines Glyphs selbst zu zeichnen, normalerweise nicht verwendet wird. Streichen Sie daher in Linie::draw die entsprechenden Zeilen und setzen Sie in Glyphtest.c das Objekt "linie" analog zum Vorgehen in Abschnitt 1.3 in ein Background-Objekt. Wenn Sie als Hintergrundfarbe style–>background() verwenden, entspricht die Hintergrundfarbe Ihres Fensters automatisch der Voreinstellung der anderen Bildschirmfenster. Bitte denken Sie auch daran, das erforderliche Headerfile background.h in header.h zu ergänzen.

Als nächstes könnte der Konstruktor der Klasse Linie parametrisiert werden. Als Argumente können z.B. die acht Parameter angegeben werden, die in Linie::request zum Aufbau des Requisition-Objekts verwendet werden. Diese Argumente sollten im Konstruktor Linie::Linie in neu einzuführende private Membervariablen abgespeichert werden. Die Memberfunktion Linie::request kann dann diese Variablen verwenden. Nun können Sie in Glyphtest.c bequem die Größe des erzeugten Linie-Objekts steuern. Probieren Sie unterschiedliche Parametersätze aus und beobachten Sie, wie sich das Verhalten des Bildschirmfesters ändert, wenn Sie eine andere natürliche Größe oder Dehnbarkeit bzw. Stauchbarkeit angeben.

Der dritte Vorschlag betrifft die Vereinfachung der Zeichenroutinen. Wir wollen uns nun die Arbeit durch die Verwendung der 2.6-kompatiblen Klasse Painter vereinfachen. Die Vor- und Nachteile der Verwendung dieser Klasse wurden bereits in Abschnitt 2.2 diskutiert. Sie werden aber sehen, daß diese Klasse unsere Arbeit erheblich vereinfacht:

```
class Painter
  {
    public:
      Painter();
      void SetColors(const Color *fg, const Color *bg);
      void SetBrush(const Brush*);
      const Brush* GetBrush() const;
      void SetFont(const Font*);
      const Font* GetFont() const;
      virtual void ClearRect(Canvas*, Coord x1, Coord y1,
                             Coord x2, Coord y2);
      virtual void Line(Canvas*, Coord x1, Coord y1,
                        Coord x2, Coord y2);
      virtual void Circle(Canvas*,Coord x, Coord y, int r);
      virtual void BSpline(Canvas*, Coord *x, Coord *y, int n);
      virtual void Text(Canvas*, const char *str,
                        Coord x, Coord y);
      ...
  };
```

Jedes Painter-Objekt besitzt einen bestimmten "Graphikstatus", der z.B. festlegt, mit welchem Pinsel gezeichnet oder mit welchem Zeichensatz Text ausgegeben wird. Der Standardkonstruktor erzeugt für das Painter-Objekt automatisch einen definierten Graphikstatus, der unseren Anforderungen für dieses Beispiel genügt.

Die einzelnen Attribute des Graphikstatus können durch Operationen wie SetBrush bzw. GetBrush neu gesetzt bzw. abgefragt werden. Mit SetColors werden Vorder- und Hintergrundfarbe gesetzt. Die Hintergrundfarbe wird beispielsweise beim Löschen eines rechteckigen Bildschirmbereiches mittels ClearRect verwendet, die Vordergrundfarbe z.B. für das Zeichnen einer Geraden. Am Beispiel der Routine Painter::Line sehen Sie, wie einfach nun Zeichenoperationen ablaufen. Sie brauchen nur den zu "bemalenden" Canvas und die Koordinaten der Endpunkte anzugeben und erhalten so die gewünschte Linie. Aber auch sehr komplexe Zeichenoperationen wie die Darstellung von Kreisen (Painter::Circle) oder Splines (Painter::Spline) ist möglich. Ebenso kann mittels Painter::Text Text ausgegeben werden. Eine genauere Erklärung der Parameter und weiterer Memberfunktionen finden Sie im Referenzmanual [LIN 91].

In dem folgenden Listing sehen Sie, wie die Routine Linie::draw für die Verwendung eines Painter-Objekts modifiziert werden muß:

```
void Linie::draw(Canvas *c,const Allocation &a) const
{
  Painter *painter = new Painter();

  Coord x1 = a.left();
  Coord y1 = a.bottom();
  Coord x2 = a.right();
  Coord y2 = a.top();

  painter->Line(c,x1,y1,x2,y2);
}
```

Der hier beschrittene Weg, sich auf die Voreinstellung des Graphikstatus zu verlassen, kann natürlich gefährlich sein. Wollen Sie z.B. Vorder- und Hintergrundfarbe explizit setzen, so hat Linie::draw diese Gestalt:

```
void Linie::draw(Canvas *c,const Allocation &a) const
{
  Session *session = Session::instance();
  Style *style = session->default_display()->style();

  Painter *painter = new Painter();
  painter->SetColors(style->foreground(),
                     style->background());

  Coord x1 = a.left();
  Coord y1 = a.bottom();
  Coord x2 = a.right();
  Coord y2 = a.top();
```

```
  painter->Line(c,x1,y1,x2,y2);
}
```

Um für Vorder- und Hintergrundfarbe wirklich die auf dem Bildschirm üblichen Standardeinstellungen zu erhalten, wird auf den Style des Defaultbildschirms zugegriffen. Dazu wird ein Zeiger auf *das* Session-Objekt der InterViews-Anwendung benötigt. Diesen erhält man durch die bisher noch nicht erklärte statische Memberfunktion Session::instance().

Damit Sie die Klasse Painter überhaupt verwenden können, müssen Sie das zugehörige Headerfile in header.h aufführen. Die Datei header.h sieht dann folgendermaßen aus:

```
#ifndef header_h
#define header_h

#include <InterViews/session.h>
#include <InterViews/display.h>
#include <InterViews/style.h>
#include <InterViews/window.h>
#include <InterViews/glyph.h>
#include <InterViews/background.h>   // neu
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <InterViews/painter.h>      // neu
#include <iostream.h>

#include "Linie.h"

#endif
```

Außerdem müssen Sie dem Compiler noch mitteilen daß Sie jetzt "2.6-kompatibel" kompilieren wollen. Dies geschieht durch Angabe des Makros "Use_2_6()" im Imakefile:

```
SPECIAL_IMAKEFLAGS = -DUseInstalled -DTurnOptimizingOn=0

SRCS = \
    Linie.$(CCSUFFIX) \
    glyphtest.$(CCSUFFIX)

OBJS = \
    Linie.o \
    glyphtest.o
```

```
AOUT = glyphtest

Use_libInterViews()
Use_2_6()

ComplexProgramTarget(glyphtest)

MakeObjectRule(Linie,Linie,NullParameter)
MakeObjectRule(glyphtest,glyphtest,NullParameter)
```

Erzeugen Sie daraus wiederum mittels "ivmkmf" ein Makefile und probieren Sie das veränderte Programm aus. Schließlich können Sie noch versuchen, Kreise oder Splines auf den Bildschirm zu bringen.

Kapitel 3

Eingabe

Nachdem Sie im letzten Kapitel die Grundlagen der Bildschirmausgabe kennengelernt haben, soll nun die Behandlung von Benutzereingaben besprochen werden. Moderne Graphikoberflächen sind interaktiv, d.h. Maus- oder Tastatureingaben werden bearbeitet und die resultierenden Änderungen sofort am Bildschirm angezeigt.

Die Struktur dieses Kapitels gleicht der des Kapitels 2. Nach der Definition einer kleinen Aufgabenstellung und der Diskussion der benötigten InterViews-Klassen schließt sich die Implementierung der Aufgabenstellung an.

3.1 Aufgabenstellung

Um das Beispielprogramm so knapp und leicht verständlich wie möglich zu halten, wird hier keine völlig neue Aufgabe gestellt. Vielmehr wird das Beispiel des vorhergehenden Kapitels um eine einfache Eingabemöglichkeit erweitert. Bisher wird in dem Bildschirmfenster nur eine diagonale schwarze Linie dargestellt. Das Programm soll nun durch Drücken einer der drei Tasten "d", "s" oder "w" dazu veranlaßt werden, eine diagonale, senkrechte oder waagrechte Linie zu zeichnen.

3.2 Notwendige InterViews-Klassen

In diesem Abschnitt werden Sie die drei Klassen Event, Handler und Hit kennenlernen. Außerdem wird die Klasse Glyph eingehender als bisher behandelt.

Event:

Objekte dieser Klasse repräsentieren Eingabeereignisse wie z.B. Maus- oder Tastaturklicks. Der für uns wesentliche Deklarationsausschnitt sieht folgendermaßen aus:

```
typedef unsigned int EventType;
typedef unsigned int EventButton;

class Event
  {
    public:
      enum {undefined, motion, down, up, key, other_event};
      enum {none, any, left, middle, right, other_button};
      virtual EventType type() const;
      virtual Coord pointer_x() const;
      virtual Coord pointer_y() const;
      virtual EventButton pointer_button() const;
      virtual unsigned char keycode() const;
      virtual unsigned int mapkey(char* buffer,
                                  unsigned int len) const;
      ...
  };
```

Der InterViews-Programmierer braucht sich um Verwaltungsaufgaben wie Erzeugen oder Löschen von Event-Objekten nicht zu kümmern. Diese werden vom System erledigt. Für jedes Display, das eine InterViews-Applikation geöffnet hat (in der Regel nur das Defaultdisplay), verwaltet das System einen Strom von Events. Tritt nun z.B. ein Mausklick auf, so wird das entsprechende Event automatisch erzeugt und in den Strom eingefügt. Gleichzeitig versucht das System, die Events dieses Stroms der InterViews-Anwendung zur Verarbeitung zu übergeben. Die Übergabe und die Verarbeitung der Events werden bei der Diskussion der Klassen Glyph und Handler besprochen. Zunächst soll aber die Klasse Event anhand ihrer Deklaration genauer untersucht werden.

Zu Beginn der Deklaration werden die beiden Aufzählungstypen EventType und EventButton definiert. Ein motion-Event wird erzeugt, wenn der Mauszeiger auf dem Bildschirm bewegt wird. Ein down- bzw. up-Event entsteht beim Drücken bzw. Loslassen eines Mausknopfs. Ein key-Event tritt auf, wenn auf der Tastatur eine Taste betätigt wird. Alle anderen Events werden als "other_event" bezeichnet und sind für uns nicht interessant. Dazu gehören z.B. Events, die zur Fensterverwaltung unter X nötig sind. Elemente des Typs EventButton spezifizieren, ob eine Maustaste bzw. welche Maustaste gedrückt wurde. Event::left zeigt beispielsweise an, daß die linke Maustaste betätigt wurde. Die Bedeutung der restlichen Elemente dieses Aufzählungstyps ist offensichtlich.

Die Memberfunktion type liefert den Typ des Events zurück. pointer_x und pointer_y geben die Position des Mauszeigers zu dem Zeitpunkt an, an dem das Event auftrat. Die Angabe der Position erfolgt relativ zur linken unteren Ecke des Fensters, zu dem das Event gehört. Die Routine pointer_button gibt an, ob ein Mausknopf bzw. welcher Mausknopf gedrückt wurde. Ist das Event vom Typ key, so liefert keycode den maschinenspezifischen Tastaturcode der gedrückten Taste. Um die Maschinenabhängigkeit auszuschalten, wird die Routine mapkey bereitgestellt. Sie übersetzt den intern gespeicherten Tastaturcode in einen String. Dazu werden ihr als erstes Argument die Adresse eines Buffers und als zweites Argument dessen Länge übergeben. Als Resultat liefert mapkey die Länge des Strings zurück, den sie im Buffer abgelegt hat. Normalerweise ist diese Länge 1 und in buffer[0] findet sich dann das Zeichen, das zu der gedrückten Taste gehört.

Glyph:

Als nächstes wird gezeigt, wie das System Events an eine InterViews-Applikation übergibt. Wie bereits mehrfach erwähnt sind Glyphs die Basisobjekte für die Konstruktion von InterViews-Graphikoberflächen. Sie sind nicht nur für die Ausgabe auf den Bildschirm zuständig, sondern dienen auch der Verarbeitung von Eingabeereignissen. Wenn während einer InterViews-Anwendung ein Event auftritt, wird dieses in den oben beschriebenen Eventstrom eingefügt. Zugleich versucht das System, die Events dieses Stroms an die InterViews-Applikation weiterzuleiten. Dazu wird zunächst das zuständige Bildschirmfenster bestimmt (ein Fenster wird in InterViews durch die Klasse Window bzw. durch davon abgeleitete Klassen repräsentiert, vgl. Kapitel 1). Sie wissen ja bereits, daß ein Window-Objekt im Konstruktor einen Zeiger auf ein Glyph-Objekt erhält, das das Aussehen und Verhalten des Fensters bestimmt. Da das Fenster das vom System übergebene Event nicht interpretieren kann, leitet es das Ereignis an seinen Glyph weiter. Bis zu diesem Punkt laufen alle eben beschriebenen Aktionen automatisch ab. Unsere InterViews-Applikation wird erst dann wieder aktiv, wenn dem Glyph das Event übergeben wird. Diese Übergabe erfolgt durch Aufruf der Memberfunktion Glyph::pick, deren Schnittstelle folgende Gestalt hat:

```
class Glyph : virtual public Resource
  {
    public:
      virtual void pick(Canvas*,const Allocation&,
                        int depth, Hit&);
      ...
  };
```

Pick ist neben request, allocate und draw die wichtigste Memberfunktion von Glyph. Wie diese drei Funktionen ist auch pick standardmäßig mit leerem Rumpf vordefiniert und muß vom Programmierer neu implementiert werden. Da Glyphs auch zusammengesetzt werden können, kann pick in unterschiedlichen Zusammenhängen aufgerufen werden. Zusammengesetzte Glyphs werden in Kapitel 4 und vor allem in Kapitel 5 noch sehr ausführlich behandelt. An dieser Stelle soll zu Ihrem Verständnis nur das Wichtigste zu diesem Thema erwähnt werden:

Es gibt primitive Glyphs, MonoGlyphs und zusammengesetzte Glyphs. Primitive Glyphs (z.B. Labels) enthalten keine weiteren Glyphs, während MonoGlyphs genau einen Glyph enthalten. Zusammengesetzte Glyphs können mehrere Glyphs umfassen. Beispiele für zusammengesetzte Glyphs sind Objekte vom Typ LRBox, die ihre Komponenten von links nach rechts anordnen.

Mehrfach wurde schon erklärt, daß ein Window einen Zeiger auf einen Glyph erhält. In Kapitel 2 wurde ein primitiver Glyph, nämlich ein Objekt unserer Klasse Linie, verwendet. In der Regel wird hier aber nur der Zeiger auf die Wurzel einer Glyphhierarchie übergeben, die aus primitiven Glyphs, MonoGlyphs und zusammengesetzten Glyphs aufgebaut ist. Das System ruft in diesem Fall Glyph::pick auf dieser Wurzel auf. Die pick-Routinen der MonoGlyphs und der zusammengesetzten Glyphs rufen lediglich die pick-Routinen ihrer Komponenten auf, bis die Blätter der Glyphhierarchie, die primitiven Glyphs, erreicht sind. Mit der genauen Implementierung von pick bei MonoGlyphs und zusammengesetzten Glyphs brauchen wir uns nicht zu beschäftigen, da die InterViews-Klassenbibliothek bereits genügend solche Glyphs enthält. Wir werden nur primitive Glyphs programmieren und uns mit der pick-Implementierung für diesen Fall eingehender auseinandersetzen.

Dieser Überblick auf die grundlegenden Eigenschaften von Glyphs war leider nötig, damit Sie die Parameter der pick-Routine verstehen können. Die beiden ersten Parameter "Canvas*" und "const Allocation&" haben dieselbe Bedeutung wie bei den beiden Memberfunktionen allocate und draw. "Canvas*" repräsentiert die Leinwand, auf die der Glyph zeichnen soll. Die Referenz auf das Allocation-Objekt enthält den Koordinatenbereich, den der Glyph zur Bildschirmausgabe verwenden darf. Der dritte Parameter "int depth" zeigt dem Glyph an, auf welcher Stufe der Glyphhierarchie er sich befindet. Wurde beispielsweise ein Bildschirmfenster durch

```
Window *w = new ApplicationWindow(new Linie());
```

erzeugt, so wird Linie::pick vom System mit depth=0 aufgerufen. Entstand das Fenster dagegen mittels

```
Window *w = new ApplicationWindow(new LRBox(new Linie(),...));
```

so wird Linie::pick von LRBox::pick mit depth=1 aufgerufen. Der vierte Parameter "Hit&" von pick wird im nächsten Teilabschnitt erklärt.

Hit:

Bisher wurde immer davon gesprochen, daß einem Glyph durch Aufruf der Memberfunktion pick ein Event zur weiteren Verarbeitung übergeben wird. Glyph::pick hat aber keinen Parameter vom Typ Event. Das Event wird in dem Parameter "Hit&" an pick übertragen. Glyph::pick entnimmt dem Hit-Objekt die Eventinformation, wertet sie aus und trägt in das Hit-Objekt Informationen ein, die dem System mitteilen, ob bzw. wie der Glyph auf das Event reagieren will. Das Hit-Objekt wird demzufolge als transienter Parameter verwendet. Im weiteren wird die Klasse Hit genauer beschrieben:

```
class Hit
  {
    public:
      Hit(const Event*);
      Hit(Coord left, Coord bottom, Coord right, Coord top);
      virtual const Event* event() const;
      virtual void target(int depth, Glyph*,
                          GlyphIndex, Handler* = nil);
      ...
  };
```

In unseren Beispielen werden wir Hit-Objekte nicht selbst erzeugen. Trotzdem soll kurz auf die beiden abgebildeten Konstruktoren eingegangen werden. Der erste ist klar, er erhält als Parameter einen Zeiger auf das Event, mit dem das Hit-Objekt konstruiert werden soll. Darüberhinaus ist es aber auch möglich, ein Hit-Objekt mit einem Rechteck zu erzeugen. Die pick-Routine wird nämlich nicht nur vom System zur Übermittlung eines Events aufgerufen, sondern auch vom Programmierer selbst, um feststellen zu lassen, ob ein Glyph eine bestimmte Rechtecksfläche schneidet. Diese beiden Aufgaben sind sehr ähnlich und können mit den entsprechenden Fallunterscheidungen leicht gemeinsam bei der Implementierung der pick-Routine realisiert werden. Wir werden allerdings nur die Verarbeitung von Events programmieren.

Die Memberfunktion event liefert einen Zeiger auf das Event, mit dem das Hit-Objekt erzeugt wurde oder nil, falls der Rechteck-Konstruktor verwendet wurde.

Mittels der Routine target kann ein Glyph Eintragungen in einem Hit-Objekt vornehmen. Wurde das Hit-Objekt mit einem Event konstruiert, so wird dem System auf diesem Weg mitgeteilt, daß der Glyph auf das Event reagieren will. Der erste Parameter von target gibt die Hierarchiestufe des eintragenden Glyphs an. Er wird im Regelfall direkt aus dem Argument depth von Glyph::pick übernommen. Als zweites Argument wird ein Zeiger auf den Glyph selbst (this) angegeben. Damit erklärt der Glyph, daß er an dem Event interessiert ist. Als drittes Argument wird ein Index angegeben, bei primitiven Glyphs 0. Als viertes Argument wird ein Zeiger auf ein Handler-Objekt übergeben. Dieses Handler-Objekt legt fest, welche Aktionen als Reaktion auf das Event ausgeführt werden sollen. Natürlich könnten diese Aktionen gleich in Glyph::pick angegeben werden. Dennoch sollten zu diesem Zweck Handler verwendet werden, weil dieser Mechanismus in InterViews Vorteile bietet. So ist es beispielsweise möglich, daß mehrere Glyphs einer Glyphhierarchie auf ein Event reagieren wollen und ein entsprechendes Handler-Objekt in das Hit-Objekt eintragen. Das System kann nun daraus entsprechend bestimmten Regeln einen geeigneten Handler auswählen und ausführen.

Wird dagegen ein Hit-Objekt mittels des Rechteck-Konstruktors erzeugt, so teilt der Glyph durch Hit::target mit, daß er beim Zeichnen die angegebene Rechtecksfläche überschneiden würde. So erhält die Routine, die Glyph::pick auf der Wurzel einer Glyphhierarchie aufruft, in dem Hit-Objekt eine Hierarchie von Glyphs, die das Rechteck zumindest teilweise überdecken. In diesem Fall braucht in Hit::target natürlich kein Handler angegeben werden. Die übrigen Parameter von target haben eine ähnliche Bedeutung wie bei einem Hit-Objekt, das mit einem Event erzeugt wurde.

Handler:

Zum Abschluß wird noch die leicht verständliche Klasse Handler erklärt. Ihre Deklaration fällt sehr kurz aus:

```
class Handler : virtual public Resource
  {
    public:
      virtual void event(Event&) = 0;
  };
```

Die rein virtuelle Funktion event kennzeichnet Handler als abstrakte Basisklasse. Der InterViews-Programmierer muß eigene Subklassen von Handler ableiten. Damit legt diese Klasse lediglich ein Protokoll fest. In Glyph::pick trägt der Glyph mittels Hit::target einen Zeiger auf ein Handler-Objekt in dem Hit-Objekt ein. Das System ruft daraufhin automatisch Handler::event auf diesem Handler-Objekt auf. Demzufolge müssen also in Handler::event die Aktionen festgelegt werden, mit denen der Glyph auf ein Event reagieren will.

Zusammenfassung:

Zum Abschluß dieses Abschnitts möchte ich noch einmal die Schritte zusammenfassen, die beim Auftreten eines Eingabeereignisses in einer InterViews-Applikation ablaufen:

1. Das System bestimmt das Fenster, zu dem das Ereignis gehört.

2. Das System ruft auf dem Wurzelglyph des Fensters Glyph::pick mit einem Hit-Objekt auf, das mit dem entsprechenden Event erzeugt wurde.

3. Die Glyphhierarchie wird mit sukzessiven pick-Aufrufen durchlaufen. Dabei werden evtl. mehrere Handler-Objekte in das Hit-Objekt eingetragen.

4. Das System wählt einen geeigneten Handler aus dem Hit-Objekt aus und führt darauf Handler::event aus.

3.3 Implementierung der Aufgabenstellung

Auch bei diesem Beispiel werden Sie feststellen, daß die eigentliche Implementierung wesentlich einfacher als erwartet ist. Die theoretischen Teile fallen in diesem Tutorial deshalb so umfangreich aus, weil Sie nicht nur die Beispiele verstehen sollen, sondern nach dem Durcharbeiten auch in der Lage sein sollen, eigene InterViews-Programme zu schreiben.

Der Code ist in die vier Files header.h, Linie.h, Linie.c und picktest.c aufgeteilt. Die Datei header.h wurde um wenige Includedateien erweitert:

```
#ifndef header_h
#define header_h

#include <InterViews/session.h>
#include <InterViews/display.h>
#include <InterViews/style.h>
#include <InterViews/window.h>
#include <InterViews/background.h>
#include <InterViews/glyph.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <InterViews/event.h>
#include <InterViews/handler.h>
#include <InterViews/hit.h>

#include <iostream.h>
#include <strings.h>

#include "Linie.h"

#endif
```

Das Hauptprogramm picktest.c unterscheidet sich nur minimal von glyphtest.c aus dem Abschnitt 2.3. Analog zum Abschnitt 2.4 wird das Linie-Objekt nicht mehr direkt in das ApplicationWindow eingesetzt, sondern vorher in einen Background-MonoGlyph gepackt. Damit entfällt auch in Linie::draw das explizite Zeichnen des Hintergrunds (siehe unten). Dementsprechend hat picktest.c nun folgende Gestalt:

```
#include "header.h"

int main(int argc,char **argv)
{
  Session *session = new Session("PickExample",argc,argv);
  Style *style = session->style();

  Linie *linie = new Linie();
  Background *bg = new Background(linie,style->background());

  Window *window = new ApplicationWindow(bg);

  return session->run_window(window);
}
```

Die wichtigen Änderungen beginnen in Linie.h, das folgende Deklarationen enthält:

```
#ifndef Linie_h
#define Linie_h

const int buflen = 20;

class Linie : public Glyph, public Handler
{
  public:
    Linie();
    virtual void request(Requisition&) const;
    virtual void allocate(Canvas*,const Allocation&,
                              Extension&);
    virtual void draw(Canvas*,const Allocation&) const;
    virtual void pick(Canvas*,const Allocation&,
                      int depth,Hit&);
    virtual void event(Event&);

  private:
    char linientype;
    Canvas *canvas;
    Allocation alloc;
    char buffer[buflen];
};

#endif
```

Die Klasse Linie ist jetzt von Glyph *und* Handler abgeleitet. Durch diesen Trick ersparen wir uns die Ableitung einer eigenen Handler-Klasse. Dadurch wird event eine Memberfunktion von Linie. Der Vorteil dieses Vorgehens liegt darin, daß event nun Zugang zu den privaten Membern von Linie hat. Damit wird die lästige Definition von Linie::set_xyz und Linie::get_xyz Memberfunktionen vermieden. Außerdem führt dieses Verfahren zu kompakten und übersichtlichen Programmen.

In Linie.h werden vier private Membervariablen für die Klasse Linie eingeführt. Die Charactervariable linientype gibt an, ob eine diagonale, waagrechte oder senkrechte Linie gezeichnet werden soll. "Canvas *canvas" und "Allocation alloc" dienen zur Speicherung der Argumente von Linie::allocate, weil diese Informationen später noch gebraucht werden. Der String buffer mit der konstanten Länge buflen wird als Zwischenspeicher benötigt, um bei einem Keyevent die Stringrepräsentation der gedrückten Taste zu errechnen.

Die Implementation der Klasse Linie ist wiederum in der Datei Linie.c enthalten:

```
#include "header.h"


Linie::Linie() : Glyph(),Handler()
{
  linientype = 'd';
  canvas = nil;
}


void Linie::request(Requisition &r) const
{
  Requirement rx(250.0,150.0,100.0,0.0);
  Requirement ry(150.0,50.0,0.0,0.0);

  r.require(Dimension_X, rx);
  r.require(Dimension_Y, ry);
}


void Linie::allocate(Canvas *c,const Allocation &a,
                                Extension &ext)
{
  ext.xy_extents(a);

  canvas = c;
```

```
  alloc.allot(Dimension_X,a.allotment(Dimension_X));
  alloc.allot(Dimension_Y,a.allotment(Dimension_Y));
}


void Linie::draw(Canvas*,const Allocation&) const
{
  Color *black = new Color(0.0,0.0,0.0);
  Brush *brush = new Brush(1.0);

  Coord x1 = alloc.left();
  Coord y1 = alloc.bottom();
  Coord x2 = alloc.right();
  Coord y2 = alloc.top();

  canvas->new_path();  // diagonale, waagrechte oder
                       // senkrechte Linie zeichnen

  if ( linientype == 'd' )
    {
      canvas->move_to(x1,y1);
      canvas->line_to(x2,y2);
    }
  else if ( linientype == 'w' )
    {
      canvas->move_to(x1,(y1+y2)/2);
      canvas->line_to(x2,(y1+y2)/2);
    }
  else if ( linientype == 's' )
    {
      canvas->move_to((x1+x2)/2,y1);
      canvas->line_to((x1+x2)/2,y2);
    }

  canvas->close_path();

  canvas->stroke(black,brush);
}


void Linie::pick(Canvas*,const Allocation&,int depth,Hit &h)
{
```

```
  if ( h.event() != nil && h.event()->type() == Event::key )
      h.target(depth,this,0,this);
}


void Linie::event(Event &e)
{
  if ( e.type() == Event::key )
    {
      int l = e.mapkey(buffer,buflen);

      if ( buffer[0] == 'd' || buffer[0] == 'w'
                            || buffer[0] == 's' )
        {
          linientype = buffer[0];

          if ( canvas != nil )
            canvas->damage(alloc.left(),alloc.bottom(),
                           alloc.right(),alloc.top());
        }
      else if ( buffer[0] == 'q' )
          Session::instance()->quit();
    }
}
```

Im Konstruktor werden die Membervariablen linientype und canvas vorbesetzt. Durch linientype = 'd' wird die diagonale Linie als Default gesetzt. Linie::request bleibt gegenüber Abschnitt 2.3 unverändert. Linie::allocate speichert nun zusätzlich die vom System übergebenen Parameter "Canvas*" und "const Allocation&" in den Membervariablen canvas und alloc ab. In Linie::draw fällt der Programmabschnitt zum Zeichnen des Hintergrunds einschließlich der Erzeugung der Farbe Weiß weg. Wie schon in Abschnitt 2.3 werden (x1,y1) bzw. (x2,y2) mit den Koordinaten der linken unteren bzw. rechten oberen Ecke besetzt. Der Abschnitt zur Erzeugung der Linie wird gemäß der Aufgabenstellung erweitert. Abhängig vom Inhalt der Membervariablen linientype wird eine diagonale, waagrechte oder senkrechte Linie gezeichnet. Die diagonale Linie erstreckt sich wie bisher von links unten nach rechts oben, die waagrechte von links mitte bis rechts mitte und die senkrechte von mitte unten bis mitte oben.

Völlig neu ist Linie::pick. Die Parameter "Canvas*" und "const Allocation&" interessieren uns nicht mehr, da sie bereits in allocate abgespeichert wurden. Im Rumpf der Routine pick wird zunächst zur Sicherheit überprüft, ob das als Referenz übergebene Hit-Objekt wirklich mit einem Event erzeugt wurde. Danach wird abgefragt, ob ein KeyEvent vorliegt, denn wir sind in unserem Beispiel ja nur an bestimmten Tastaturereignissen interessiert. Falls beide Abfragen true liefern, wird mittels h.target das Interesse des Glyphs an diesem Event ausgedrückt. Der Parameter depth wird aus der Argumentleiste von Linie::pick übernommen. Das erste this kennzeichnet den Glyph, der am Event interessiert ist. Das dritte Argument "0" wird gewählt, weil Linie ein primitiver Glyph ist. Das vierte Argument trägt das Linie-Objekt als den für das Event zuständigen Handler in das Hit-Objekt ein.

Durch die Memberfunktion Linie::event wird die Handlereigenschaft der Klasse Linie implementiert. Zunächst wird sicherheitshalber überprüft, ob das als Referenz übergebene Event wirklich ein Tastaturereignis war. Wenn dies der Fall war, wird durch Event::mapkey mit Hilfe der Linie-Membervariablen buffer die gedrückte Taste ermittelt. Falls d, w oder s gedrückt wurde, wird linientype entsprechend gesetzt. Nun soll aber auch die Linie gemäß dieser Einstellung neu gezeichnet werden. Dazu darf aber nicht einfach Linie::draw aufgerufen werden. Vielmehr wird der Canvas, auf dem die Linie abgebildet wird, "beschädigt". Dies geschieht durch die bisher noch nicht beschriebene Memberfunktion Canvas::damage:

```
class Canvas
  {
    public:
      void damage(Coord, Coord, Coord, Coord);
      ...
  };
```

Die vier Parameter geben den rechteckigen zu beschädigenden Bereich an. Als Parameter wird hier einfach der gesamte Bereich angegeben, der dem Glyph in Linie::allocate zugewiesen wurde. Nur für diesen Programmabschnitt werden die beiden Membervariablen canvas und alloc gebraucht. Durch die Beschädigung des Canvas ruft das System automatisch Linie::draw auf und stellt die Änderungen sofort am Bildschirm dar.

Schließlich wird der Taste 'q' ebenfalls eine besondere Bedeutung zugewiesen. Durch die Session-Memberfunktion quit wird die in picktest.c mittels session–>run_window gestartete Abfrageschleife von Eingabeereignissen abgebrochen. Die main-Routine kann danach die return-Anweisung ausführen und somit ein reguläres Programmende ermöglichen. Damit ist der bisher nötige explizite Programmabbruch mittels <ctrl>-c nicht mehr erforderlich. Der für die Ausführung von Session::quit benötigte Zeiger auf das globale Session-Objekt wird von der in Abschnitt 2.4 bereits kurz erwähnten statischen Memberfunktion Session::instance zurückgeliefert:

```
class Session
  {
    public:
      static Session* instance();
      ...
  };
```

Die Erstellung des zugehörigen Imakefiles sollte nun keine Probleme mehr bereiten. Übersetzen Sie das Programm und versuchen Sie eventuell, weiteren Tasten eine bestimmte Bedeutung zuzuordnen. Sie könnten auch Mausereignisse abfragen und damit beispielsweise einen kleinen Editor zum Zeichnen von Linien programmieren.

Zum Abschluß dieses Kapitels möchte ich noch die von MonoGlyph abgeleitete Klasse Listener erwähnen. Diese Klasse ermöglicht es, einen Glyph bei bestimmten Eingabeereignissen mit einem bestimmten Handler zu assoziieren. Dadurch wird derselbe Effekt erzielt, den wir mit unserem Beispielprogramm auf einfache Weise erhalten haben. Bei Interesse schlagen Sie bitte im Referenzmanual [LIN 91] nach.

Kapitel 4

Menüs

Inzwischen haben Sie nicht nur gelernt, wie in InterViews Bildschirmausgaben erzeugt werden, sondern Sie haben nun auch die Grundlagen der Eingabeverarbeitung verstanden. Nun ist es aber sehr umständlich, ein Programm über Tastendruck zu steuern. Darüberhinaus ist es oftmals schwierig, sich die für die Bedienung eines komplexen Programms erforderlichen Tastenkürzel einzuprägen. Daher haben sich für die Interaktion zwischen Programm und Benutzer andere Techniken durchgesetzt. Bekanntestes Beispiel sind die Pulldownmenüs, deren Erzeugung unter InterViews Gegenstand dieses Kapitels sein soll.

4.1 Aufgabenstellung

Wie auch in den vorangegangenen Beispielen soll der Programmcode so knapp wie möglich gehalten werden. Deshalb wird hier nur eine ganz einfache Menüstruktur aufgebaut. Das Anklicken eines Menüpunkts wird lediglich die Ausgabe einer Meldung der Form "... called" bewirken.

Die genaue Struktur des Beispielmenüs ist aus Abbildung 4.1 ersichtlich. Die Menüleiste umfaßt zwei Pulldownmenüs. Das erste, das den Titel "File" trägt, beinhaltet die Menüpunkte "Copyright" und "Quit". Das Pulldownmenü "Options" besteht lediglich aus dem Eintrag "Preferences". Als Hintergrund für die Menüstruktur wurde ein Objekt unserer Klasse Linie gewählt. Normalerweise findet sich an dieser Stelle irgendeine graphische Darstellung, die durch das Menü beeinflußt werden kann. Da unser Menü aber nur Demonstrationszwecken dient, kann hier ein beliebiger "Platzhalter" gewählt werden.

4.2 Notwendige InterViews-Klassen

Bevor wir uns das eigentliche Programm ansehen, werden zunächst die Klassen ActionCallback, Kit, Menu, TBBox und Rule vorgestellt.

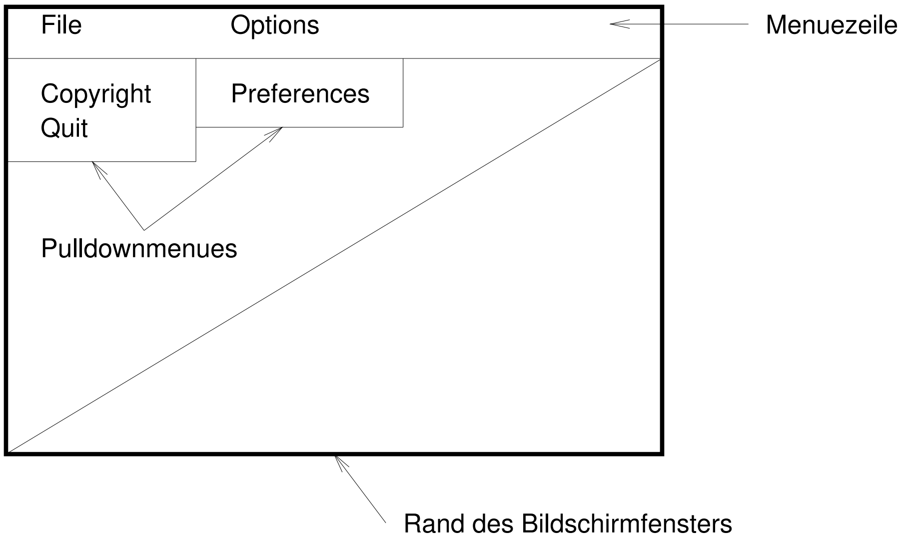

Abbildung 4.1: Menüstruktur des Beispiels

ActionCallback:

Wenn der Benutzer einen Menüeintrag anwählt, soll eine bestimmte Aktion ausgeführt werden. Eine solche Aktion wird in InterViews von einem Objekt der Klasse Action repräsentiert:

```
class Action : virtual public Resource
  {
    protected:
      Action();

    public:
      virtual void execute() = 0;
   };
```

Action ist eine abstrakte Basisklasse. Der Programmierer muß eigene Subklassen von Action ableiten. In der Implementation der virtuellen Memberfunktion execute wird dann festgelegt, was bei der Ausführung dieser Aktion geschehen soll.

Für viele Anwendungen wird beispielsweise eine Aktion benötigt, die für ein bestimmtes Objekt einer Klasse eine bestimmte Memberfunktion aufruft. Zu diesem Zweck stellt InterViews die von Action abgeleitete generische Klasse ActionCallback(Typname) zur Verfügung:

```
class ActionCallback(T) : public Action
  {
    public:
      ActionCallback(T)(T*,void (T::*)());
  };
```

Da C++ derzeit noch keine generischen Typen erlaubt, erfolgt die Deklaration und Implementierung einer Klasse ActionCallback(Typname) durch vordefinierte Makros. Sei z.B. TestMenu ein Klassenname, tm ein Zeiger auf ein Objekt vom Typ TestMenu und void TestMenu::copyright() eine Memberfunktion von TestMenu. Dann wird die Klasse ActionCallback(TestMenu) durch

```
declare(ActionCallback, TestMenu)
```

deklariert und durch

```
implement(ActionCallback, TestMenu)
```

implementiert. Mittels

```
Action *copyright = new ActionCallback(TestMenu)(tm,
                                &TestMenu::copyright);
```

wird eine Aktion copyright erzeugt, die bei

```
copyright->execute()
```

den Aufruf

```
tm->copyright()
```

bewirkt. Die Verwendung von Aktionen wird in dem Beispielprogramm im folgenden Abschnitt 4.3 vorgeführt.

Kit:

Um die Bedienung von unterschiedlichen Programmen möglichst einfach zu gestalten, wäre ein einheitliches Erscheinungsbild ("look and feel") aller graphischen Programmoberflächen wünschenswert. In der UNIX-Welt sind vor allem Motif- und OpenLook-Oberflächen weit verbreitet. Daher stellt InterViews die Klasse Kit zur Verfügung, mit der es möglich ist, interaktive Objekte wie Buttons oder Menüs mit einem bestimmten Erscheinungsbild einfach zu erzeugen. Die Deklaration der Klasse Kit hat folgende Gestalt:

```
class Kit:
  {
    public:
      static Kit* instance();
      virtual Action* quit() const;
      virtual Menu* menubar(Style*) const;
      virtual Menu* pulldown(Style*) const;
      virtual Telltale* simple_menu_item(const char*,
                                         Style*) const;
      virtual Telltale* simple_menubar_item(const char*,
                                            Style*) const;
      ...
  };
```

Jede InterViews-Anwendung darf höchstens ein Kit-Objekt erzeugen. Die statische Memberfunktion instance liefert einen Zeiger auf ein evtl. bereits vorhandenes Kit-Objekt zurück. Existierte zum Zeitpunkt des Aufrufs von instance noch kein Kit-Objekt, so wird automatisch eines erzeugt und ein Zeiger darauf zurückgegeben. Ein Objekt der Klasse Kit erzeugt standardmäßig Motif-Objekte. Es gibt auch noch eine von Kit abgeleitete Klasse für die Konstruktion von OpenLook-Elementen, die aber hier nicht weiter erwähnt werden soll.

Die Memberfunktion quit liefert einen Zeiger auf eine Aktion, die Session::instance–>quit() ausführt. Diese Aktion ist für einen Menüpunkt "Quit" gedacht.

Kit::menubar erzeugt mit Hilfe des als Zeiger übergebenen Style-Objekts eine Menüleiste. Kit::pulldown erzeugt ein Pulldownmenü. Die Funktion Kit::simple_menu_item konstruiert ein Telltale-Objekt für einen Menüeintrag mit dem übergebenen Text. Die Klasse Telltale ist von MonoGlyph abgeleitet und wird für Objekte verwendet, deren Aussehen davon abhängt, ob sie hervorgehoben, angewählt, etc. sind. Telltales eignen sich daher für Menüeinträge oder auch Buttons. Kit::simple_menubar_item erzeugt schließlich einen Telltale für einen Eintrag in eine Menüleiste.

Menu:

Objekte dieser Klasse dienen als Bausteine für Menüstrukturen. Wir erzeugen solche Objekte nicht selbst, sondern überlassen diese Aufgabe dem Kit-Objekt. Dennoch müssen wir uns mit der Klasse Menu genauer auseinandersetzen, weil sie die Memberfunktionen bereitstellt, die erforderlich sind, um die Menübausteine zusammenzusetzen:

```
class Menu : public MonoGlyph, public Handler
  {
    public:
      virtual void add_item(Telltale*,Action*);
```

```
      virtual void add_item(Telltale*,Menu*,Window* = nil);
      ...
  };
```

Von den vier add_item-Memberfunktionen sind nur die zwei wichtigsten abgebildet. Die erste Form wird dazu verwendet, einen Menüeintrag und die dazugehörige Aktion in ein Pulldownmenü einzufügen. Diese Aktion wird automatisch ausgeführt, wenn der Menüeintrag vom Benutzer angewählt wird. Die zweite Form von add_item findet dann Verwendung, wenn ein Pulldownmenü samt Titel in eine Menüleiste eingefügt werden soll. Der optionale Parameter Window* wird für unsere Zwecke nicht benötigt und auf nil belassen.

TBBox:

Wenn eine Menüleiste mit den zugehörigen Pulldownmenüs erzeugt wurde, muß sie noch mit dem Menürumpf (in unserem Fall ein Linie-Objekt) zusammengesetzt werden. Da eine Menüleiste ein Objekt der Klasse Menu ist (sehen Sie sich noch einmal die Memberfunktion menubar in der Deklaration der Klasse Kit an) und Menu von MonoGlyph abgeleitet ist, stellt auch eine Menüleiste wie unser Linie-Objekt nur einen speziellen Glyph dar. Damit stoßen wir hier erstmals auf das Problem, einen zusammmengesetzten Glyph zu erzeugen. Solche Fragestellungen werden ausführlich in Kapitel 5 behandelt. Wir verwenden an dieser Stelle eine TBBox, die ihre Komponenten von oben nach unten anordnet. Für uns ist folgender Deklarationsausschnitt interessant:

```
class TBBox : public Box
  {
    public:
      TBBox(Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil);
      append(Glyph*);
      ...
  };
```

Die Basisklasse Box ist von Glyph abgeleitet, somit ist auch eine TBBox ein Glyph. Der Konstruktor erlaubt bis zu zehn Komponenten. Weitere Komponenten können mit append eingefügt werden.

Rule:

Objekte der Klasse Rule werden in zusammengesetzten Glyphs, z.B. Boxen, dazu genützt, Komponenten optisch zu trennen. Ein Objekt der von Rule abgeleiteten

Klasse HRule, das in einer TBBox zwischen zwei Glyphs eingefügt wird, erzeugt eine waagrechte Linie zwischen diesen beiden Glyphs. Die Deklaration der Klasse HRule sieht wie folgt aus:

```
class HRule : public Rule
  {
    public:
      HRule(const Color*, Coord);
  };
```

Das erste Argument des Konstruktors gibt die Farbe, das zweite die Breite der Linie an. Analog zu HRule gibt es auch noch eine Klasse VRule, die senkrechte Linien zwischen Glyphs zeichnet und in LRBoxen verwendet wird.

4.3 Implementierung der Aufgabenstellung

Der Programmcode ist in die sechs Files header.h, Linie.h, Linie.c, TestMenu.h, TestMenu.c und menutest.c aufgeteilt. Die Datei header.h ist gegenüber Abschnitt 3.3 um die zusätzlich nötigen InterViews-Headerfiles für die Klassen Kit, Menu und Action bzw. ActionCallback sowie um unsere neue Headerdatei "TestMenu.h" erweitert worden:

```
#ifndef header_h
#define header_h

#include <InterViews/session.h>
#include <InterViews/display.h>
#include <InterViews/style.h>
#include <InterViews/window.h>
#include <InterViews/glyph.h>
#include <InterViews/background.h>
#include <InterViews/rule.h>
#include <InterViews/box.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <InterViews/event.h>
#include <InterViews/handler.h>
#include <InterViews/hit.h>
#include <IV-look/kit.h>
#include <IV-look/menu.h>
#include <InterViews/action.h>

#include <iostream.h>
```

```
#include <strings.h>

#include "Linie.h"
#include "TestMenu.h"

#endif
```

Die beiden Files Linie.h und Linie.c sind gegenüber Abschnitt 3.3 unverändert und können einfach übernommen werden. Neu dagegen ist TestMenu.h:

```
#ifndef TestMenu_h
#define TestMenu_h

class TestMenu : public TBBox
{
  public:
    TestMenu();
    Menu* CreateMenu();
    void copyright();
    void preferences();
};

declare(ActionCallback,TestMenu)

#endif
```

Auch aus diesem Listing können Sie die Methodik erkennen, mit der in InterViews programmiert wird: *Für eigene Objekte, die aus InterViews-Objekten zusammengesetzt sind oder die die Funktionalität von InterViews-Objekten erweitern, werden eigene Klassen definiert.* Weil unser Beispielmenü aus einer Menüleiste und einem Rumpf besteht, die zusammengebaut werden müssen und weil wir dieses Menü als ein Objekt haben wollen, wird TestMenu von TBBox abgeleitet. TestMenu ist also die Klasse, die für das Erzeugen des gesamten Menüs zuständig ist.

Die Memberfunktion CreateMenu unterstützt den Konstruktor der Klasse bei dem Aufbau des Menüs, indem sie die Menüleiste samt Pulldownmenüs erzeugt und zusammenfügt. TestMenu::copyright und TestMenu::preferences werden von den entsprechenden Menüeinträgen des erzeugten Menüs aufgerufen. Das Makro declare(ActionCallback, TestMenu) bewirkt die Deklaration der Klasse Action-Callback(TestMenu).

Die Implementation der Klassen TestMenu und ActionCallback(TestMenu) erfolgt in TestMenu.c:

```
#include "header.h"

implement(ActionCallback,TestMenu)


TestMenu::TestMenu() : TBBox()
{
  Session *session = Session::instance();

  append(CreateMenu());
  append(new HRule(session->style()->foreground(),1.0));
  append(new Linie());
}


Menu* TestMenu::CreateMenu()
{
  Session *session = Session::instance();
  Kit *kit = Kit::instance();
  Style *style = session->style();

  Menu *mb = kit->menubar(style);
  Menu *file = kit->pulldown(style);
  Menu *options = kit->pulldown(style);

  Action *copyright = new ActionCallback(TestMenu)(this,
                                  &TestMenu::copyright);
  Action *preferences = new ActionCallback(TestMenu)(this,
                                  &TestMenu::preferences);

  file->add_item(kit->simple_menu_item("Copyright",style),
                copyright);
  file->add_item(kit->simple_menu_item("Quit",style),
                kit->quit());

  options->add_item(kit->simple_menu_item("Preferences",
                                         style),
                   preferences);

  mb->add_item(kit->simple_menubar_item("File",style),file);
  mb->add_item(kit->simple_menubar_item("Options",style),
              options);
```

```
  return mb;
}


void TestMenu::copyright()
{
  cerr << "TestMenu::copyright called\n";
}


void TestMenu::preferences()
{
  cerr << "TestMenu::preferences called\n";
}
```

Das Makro implement(ActionCallback, TestMenu) implementiert die Klasse ActionCallback(TestMenu). In der nächsten Programmzeile beginnt die Implementierung der Klasse TestMenu.

Im Konstruktor von TestMenu wird das Menü mit dem Rumpf (ein Linie-Objekt) zusammengefügt. Dazu wird die von TBBox ererbte Memberfunktion append verwendet. Als erste Komponente wird die von CreateMenu erzeugte Menüleiste eingefügt. Die Menüleiste und der Rumpf werden durch ein HRule-Objekt der Breite 1.0 optisch voneinander getrennt. Abschließend wird das Linie-Objekt angefügt.

Die wichtigste Routine der Klasse TestMenu ist CreateMenu, die die eigentliche Konstruktion der gewünschten Menüstruktur leistet. In den ersten drei Zeilen werden Zeiger auf das globale Session-Objekt, das globale Kit-Objekt und das Standard-Style-Objekt gewonnen. In den drei folgenden Zeilen werden eine Menüleiste (mb) und zwei Pulldownmenüs (file und options) mit Hilfe des Kit-Objekts erzeugt. Danach werden die zwei Aktionen copyright und preferences kreiert, die für *dieses* TestMenu-Objekt die Memberfunktionen TestMenu::copyright bzw. TestMenu::preferences aufrufen. In der darauffolgenden Zeile wird mittels kit–>simple_menu_item der Menüeintrag "Copyright" erzeugt. Durch file–>add_item wird dieser Eintrag zusammen mit der zugehörigen Aktion copyright in das Pulldownmenü "file" eingetragen. Analog wird in der darunterliegenden Zeile ein Menüeintrag "Quit" konstruiert und zusammen mit der Aktion kit–>quit in dasselbe Pulldownmenü eingefügt.

Danach wird mittels options–>add_item(...) ein Menüeintrag "Preferences" erzeugt und mit der zugehörigen Aktion preferences in das Pulldownmenü "options" eingetragen.

In den nächsten beiden Zeilen werden die Menüleisteneinträge "File" und "Options" konstruiert und mit den dazugehörenden Pulldownmenüs file bzw. options in die Menüleiste eingefügt. Als Ergebnis liefert CreateMenu einen Zeiger auf die fertig zusammengesetzte Menüleiste.

Die zwei Memberfunktionen copyright und preferences werden beim Anklicken der entsprechenden Menüeinträge aufgerufen. Sie geben lediglich eine Kontrollmeldung auf der Standardfehlerausgabe aus. Eine sinnvollere Implementation bleibt Ihnen überlassen.

Das Hauptprogramm menutest.c sieht so aus:

```
#include "header.h"

int main(int argc,char **argv)
{
  Session *session = new Session("MenuExample",argc,argv);
  Style *style = session->style();

  Background *bg = new Background(new TestMenu,
                                  style->background());

  Window *window = new ApplicationWindow(bg);

  return session->run_window(window);
}
```

Es unterscheidet sich von picktest.c aus Abschnitt 3.3 nur dadurch, daß in das Background-Objekt kein Objekt der Klasse Linie, sondern ein Objekt der Klasse TestMenu gepackt wurde.

In Abbildung 4.2 ist als Zusammenfassung des Kapitels 4 und zu Ihrer besseren Orientierung aufgezeichnet, aus welchen InterViews-Objekten und eigenen Objekten das entstehende Bildschirmfenster aufgebaut ist, das die Menüstruktur enthält.

Schreiben Sie nun ein geeignetes Imakefile und übersetzen Sie das Programm. Wenn Sie es auf einem Farbbildschirm oder einem Monochromschirm mit Farbemulation starten, wird das Menü im Motif-Erscheinungsbild zu sehen sein. Durch Aufruf des Programms mittels "<programmname> -openlook" erzeugt InterViews automatisch ein Kit-Objekt, das OpenLook-Graphikelemente konstruiert. Wird das Programm auf einem monochromen Bildschirm gestartet oder geben Sie explizit "<programmname> -monochrome" an, so erhält Ihre Applikation ein auf monochrome Bildschirme abgestimmtes Erscheinungsbild.

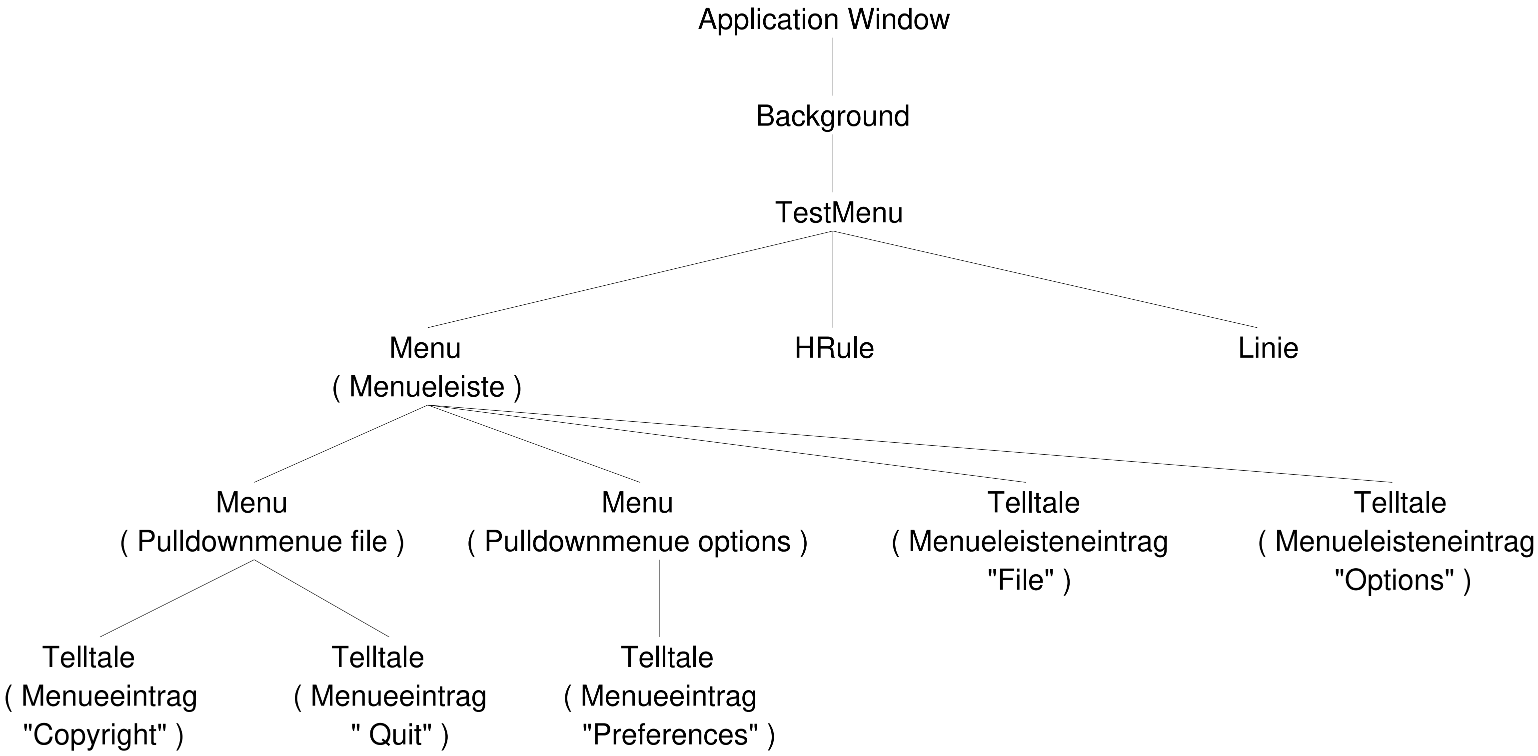

Abbildung 4.2: Zusammensetzung der Menüstruktur aus InterViews-Objekten und eigenen Objekten

Kapitel 5

Fortgeschrittene Techniken

In diesem Kapitel werden Sie lernen, wie in InterViews komplexere Benutzeroberflächen erzeugt werden. Dazu wird zu Beginn wie in den vorhergehenden Kapiteln eine Aufgabenstellung definiert, deren Implementation danach schrittweise besprochen wird.

5.1 Aufgabenstellung

Als Beispiel wird eine Dialogbox aufgebaut, die in einem eigenen Fenster auf dem Bildschirm dargestellt werden soll. Die Dialogbox wird einen kleinen Stringeditor enthalten, mit dem der Benutzer seinen Namen eingeben kann. Der Aufruf der Dialogbox wird in das Menü von Kapitel 4 eingebettet. Das Pulldownmenü "Options" erhält zu diesem Zweck den zusätzlichen Menüeintrag "DialogBox".

Grundsätzlich werden Sie bei den Erläuterungen in diesem Kapitel oft auf Punkte stoßen, die bereits früher einmal kurz erklärt wurden. Solche Wiederholungen sind beabsichtigt, weil Sie einem besseren Verständnis der oft recht komplexen Sachverhalte dienen.

5.2 Glyphs

Wie Sie bereits wissen, sind Glyphs die Basisobjekte für die Konstruktion einer InterViews-Benutzeroberfläche. Man unterscheidet drei Arten von Glyphs:

1. Primitive Glyphs wie z.B. Labels für Text haben keine Teilkomponenten und werden direkt auf dem Bildschirm abgebildet. Sie stellen einer Glyphhierarchie die Blätter dar.

2. MonoGlyphs enthalten genau einen Glyph, der auch als "Body" des MonoGlyphs bezeichnet wird. MonoGlyphs ändern das Aussehen bzw. das Verhalten ihres Bodys. So zeichnet ein Background-MonoGlyph beispielsweise

einen farbigen Hintergrund unter seinen Body. Ein Border-MonoGlyph zeichnet um seinen Body einen Rand mit variabler Breite.

Diese beiden Beispiele stehen für eine Vielzahl verfügbarer MonoGlyphs, die die unterschiedlichsten Aufgaben erfüllen können.

3. Zusammengesetzte Glyphs wie z.B. TBBoxes enthalten eine oder mehrere Glyph-Komponenten. Diese Komponenten werden auf bestimmte Weise angeordnet und auf dem Bildschirm abgebildet.

Allgemein werden InterViews-Benutzeroberflächen durch Konstruktion einer Glyphhierarchie aus zusammengesetzten Glyphs, MonoGlyphs und primitiven Glyphs aufgebaut. Die Wurzel dieser Hierarchie wird einem Window-Objekt als Argument übergeben. Bei der Darstellung des betreffenden Bildschirmfensters wird die gesamte Hierarchie dann automatisch am Bildschirm abgebildet.

Eine Glyphhierarchie ist übrigens kein Baum, sondern ein azyklischer gerichteter Graph, weil ein Knoten mehrere Vorgänger haben kann. Dies beruht auf der Tatsache, daß Glyphs "shared objects" sind. Aus diesem Grund muß beispielsweise ein Label, das in einer Dialogbox zweimal vorkommt, nur einmal erzeugt werden. Sehen wir uns zur Verdeutlichung Abbildung 5.1 an. Eine LRBox soll

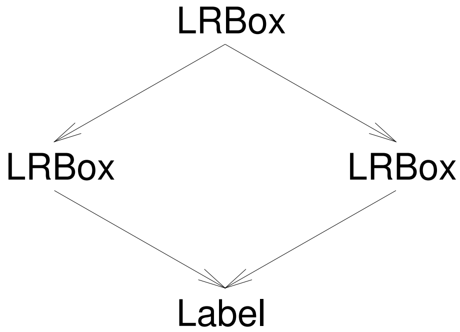

Abbildung 5.1: Beispiel einer einfachen Glyphhierarchie

wiederum zwei LRBoxen mit jeweils demselben Label enthalten. Der zughörige InterViews-Code sieht in etwa wie folgt aus:

```
Label *l = new Label("Hallo!", ...);
LRBox *b1 = new LRBox(l);
LRBox *b2 = new LRBox(l);
LRBox *b3 = new LRBox(b1, b2);
```

Wird der Zeiger auf die Wurzel dieser Hierarchie, also b3, an ein Window-Objekt übergeben und das entsprechende Fenster am Bildschirm abgebildet, so enthält das Fenster im Prinzip das in Abbildung 5.2 angegebene Bild. Die schwarzen Linien um die Boxen und die Zwischenräume dienen hier nur Ihrer Orientierung und werden von InterViews selbstverständlich nicht dargestellt.

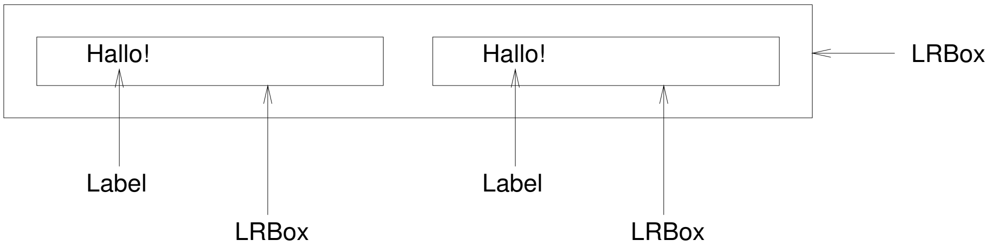

Abbildung 5.2: Stilisierte Darstellung der einfachen Glyphhierarchie

Das Label l wurde also lediglich einmal erzeugt und hat nach Abbildung 5.1 zwei Vorgänger in der Glyphhierarchie. Durch dieses Konzept der "shared objects" kann unter Umständen eine erhebliche Speicherplatzersparnis erreicht werden. Im obigen Beispiel könnten übrigens auch die beiden LRBoxes b1 und b2, die das Label enthalten, als eine einzige "shared LRBox" realisiert werden.

An dieser Stelle soll nochmals kurz auf das Prinzip der shared objects eingegangen werden. Shared objects sind alle Objekte von InterViews-Klassen oder eigenen Klassen, die Resource als Basisklasse haben. Die Klasse Glyph ist beispielsweise direkt von Resource abgeleitet. Dadurch sind auch alle Objekte von Klassen, die direkt oder indirekt von Glyph abgeleitet sind, shared objects. Beispiele dafür sind Objekte der Klassen MonoGlyph, Background, LRBox und unserer eigenen Klasse Linie.

Alle shared objects verfügen über einen Referenzzähler, der die Anzahl der Anwendungsstellen eines "shared object" angibt. Nach dem Erzeugen des Objektes enthält dieser Zähler den Anfangswert Null. Alle shared objects haben von der Basisklasse Resource die Memberfunktionen ref und unref geerbt. Durch ref wird der Zähler um eins erhöht und durch unref um eins erniedrigt. Erreicht der Zähler durch einen unref-Aufruf einen Wert kleiner gleich Null, so wird das Objekt automatisch zerstört. *Niemals darf ein shared object mit delete gelöscht werden!* Ein evtl. vorhandener Destruktor wird beim automatischen Löschen selbständig aufgerufen.

Alle InterViews-Routinen, die ein shared object als Argument haben, führen darauf ref aus. Wird dieses Argument nicht mehr benötigt, wird es mittels unref dereferenziert und damit evtl. automatisch gelöscht. Der Programmierer braucht daher für shared objects, die an InterViews-Routinen übergeben werden, nicht selbst ref oder unref aufrufen, er sollte dies sogar vermeiden.

Sie können den ref/unref-Mechanismus vielleicht auch in eigenen Klassen brauchen, wenn Sie z.B. Listen aufbauen. Die Listenelemente könnten Objekte einer von Resource abgeleiteten Klasse sein. Dadurch müssen Sie mehrmals verwendete Listenelemente nur einmal erzeugen. Die Speicherverwaltung dieser Listenelemente wird dann von der Klasse Resource erledigt.

Nach diesem Exkurs wollen wir uns aber wieder den Glyphs zuwenden. In den nachfolgenden Unterabschnitten werden die wichtigsten Vertreter der drei

Glyphsorten vorgestellt bzw. wiederholt.

5.2.1 Primitive Glyphs

Character:

Ein Character-Objekt dient der Darstellung eines Zeichens auf dem Bildschirm. Die Deklaration der Klasse hat folgende Gestalt:

```
class Character : public Glyph
  {
    public:
      Character(long code, const Font*, const Color*);
      ...
  };
```

Als erstes Argument erhält der Konstruktor den Index des darzustellenden Zeichens in dem Zeichensatz, der als zweites Argument übergeben wird. Die Berechnung dieses Index besteht lediglich aus der Konversion des entsprechenden char-Wertes in einen long-Wert. So wird z.B. der Character 'a' durch Angabe von

```
code = (long)'a';
Character *c = new Character(code, ...);
```

erzeugt. Daneben verlangt der Kontruktor noch die Angabe der Farbe, in der der Character gezeichnet werden soll.

Label:

Ein Label bietet die Möglichkeit, eine Sequenz von Characters durch ein Objekt darzustellen. Die Klassendeklaration ist selbsterklärend:

```
class Label : public Glyph
  {
    public:
      Label(const char*, const Font*, const Color*);
      ...
  };
```

An dieser Stelle möchte ich darauf hinweisen, daß die Klasse Kit für etliche Glyphs, z.B. Labels eine Memberfunktion bereitstellt. Wollen Sie beispielsweise Labels mit einem bestimmten Erscheinungsbild (z.B. Motif) darstellen, so sollten Sie das Label mittels Kit::label erzeugen. Die Klasse Kit ist zur Zeit (InterViews 3.0.1) leider noch nicht vollständig. Wenn Sie ein bestimmtes Graphikobjekt erzeugen wollen, sollten Sie sich aber dennoch zuvor die Deklaration von Kit ansehen, ob nicht eine Memberfunktion für dieses Objekt zur Verfügung steht.

Rule:

Objekte dieser Klasse dienen zur optischen Trennung von Komponenten zusammengesetzter Glyphs durch waagrechte oder senkrechte Linien. Die Klassendeklaration hat folgende Gestalt:

```
class Rule : public Glyph
  {
    public:
      Rule(DimensionName DN, const Color*, Coord thickness);
      ...
  };
```

DN = Dimension_X führt zu einer waagrechten Linie und wird zur Trennung von Komponenten in TBBoxes verwendet. DN = Dimension_Y erzeugt analog eine senkrechte Linie und wird in LRBoxes benützt. Zur Vereinfachung wurden die Klassen HRule und VRule abgeleitet, deren Deklarartion selbsterklärend ist:

```
class HRule : public Rule
  {
    public:
      HRule(const Color*, Coord thickness);
      ...
  };

class VRule : public Rule
  {
    public:
      VRule(const Color*, Coord thickness);
      ...
  };
```

Glue:

Objekte dieser Klasse dienen als "Klebstoff" (bessere Übersetzung: "Feder") zwischen den Komponenten zusammengesetzter Glyphs. Glues werden auf dem Bildschirm als Zwischenräume variabler Größe dargestellt. Ein Glue-Objekt hat eine natürliche Größe und kann in bestimmten Schranken vergrößert bzw. verkleinert werden.

Vielleicht erinnern Sie sich noch an das Beispiel aus Kapitel 1. Dort wurde ein Label (nur durch ein Background-Objekt hinterlegt) direkt in das Fenster eingefügt. Als Ergebnis erhielten Sie ein winziges Fenster, in das gerade der Text paßte und dessen Größe nicht veränderbar war. Die Aufgabe aus Kapitel 1 kann aber wesentlich eleganter gelöst werden, indem Sie in eine LRBox einen "HGlue" (H steht für horizontal, wird aber gleich genauer erklärt), das Label und nochmals einen "HGlue" einfügen. Wenn Sie den beiden HGlues die gleiche natürliche Größe $n > 0$ geben, so befindet sich das Label zentriert im Fenster mit freiem Raum auf beiden Seiten. Wenn beide HGlues die gleiche Vergrößerbarkeit g erhalten, kann das Fenster (genauer: die LRBox, die das Label und die HGlues enthält) um maximal $2 \times g$ in x-Richtung (horizontal) vergrößert werden, wobei das Label automatisch zentriert wird. Wenn dagegen für den linken HGlue eine kleiner Vergrößerbarkeit als für den rechten angegeben wird, erscheint das Label bei einem vergrößerten Fenster näher am linken als am rechten Rand der LRBox. Bei der Layoutberechnung wird also auch berücksichtigt, ob innerhalb einer Box ein bestimmter Glue evtl. stärker als ein anderer gedehnt werden kann. Ähnliche Regeln gelten auch für die Verkleinerbarkeit von Glues.

Wenn Sie zusätzlich noch über und unter dem Label Freiraum schaffen wollen, müssen Sie eine TBBox erzeugen, die einen VGlue, die eben beschriebene LR-Box und einen weitern VGlue enthält. Durch geeignete Angabe der natürlichen Größe, der Vergrößerbarkeit und der Verkleinerbarkeit kann die vertikale Position der LRBox (und somit des Labels) bei normalem, vergrößertem und verkleinertem Fenster beeinflußt werden. Die Struktur der resultierenden TBBox ist in Abbildung 5.3 nochmals anschaulich dargestellt.

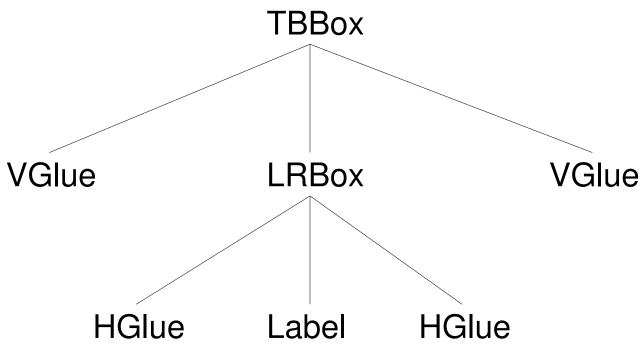

Abbildung 5.3: Struktur eines zusammengesetzten Glyphs mit Glues

Glues werden aber nicht nur verwendet, um die Fenstergröße variabel zu halten. Diese Aufgabe kann oft durch entsprechende Angaben für die Vergrößerbarkeit und Verkleinerbarkeit eines Glyphs gelöst werden, wie dies in Kapitel 2 für die von Glyph abgeleitete Klasse Linie vorgeführt wurde.

Eine zweite Klasse von Layoutproblemen ergibt sich, wenn z.B. zwei TBBoxen unterschiedlicher Höhe zu einer LRBox zusammenzufassen sind. Dieses Problem läßt sich mit Hilfe von Glues lösen: Man braucht nur eine weitere TBBox erzeugen, in die ein VGlue, die kleinere der beiden ursprünglichen TBBoxen und ein zweiter VGlue eingefügt werden. Wenn für die beiden VGlues die natürliche Größe, die Vergrößerbarkeit und die Verkleinerbarkeit so gewählt werden, daß unter Ausnutzung dieser Angaben die neue TBBox so groß wie die größere der beiden ursprünglichen Boxen gemacht werden kann, kann InterViews diese beiden Boxen problemlos zu einer LRBox zusammenfügen. Werden für beide VGlues dieselben Parameter benützt, so erscheint zusätzlich die kleinere der beiden ursprünglichen TBBoxen zentriert zur größeren.

Wenn Sie das Satzsystem TEX kennen, werden Sie feststellen, daß dort für die Layoutberechnung ebenfalls dieses Boxprinzip verwendet wird. Dies ist nicht weiter verwunderlich, weil InterViews an der Stanford University entwickelt wird, wo auch Donald Knuth, der Autor von TEX, arbeitet. Die InterViews-Autoren haben das Boxprinzip von Knuths Arbeit weitgehend übernommen (vgl. auch [KNU 86]).

Nun wollen wir aber die Klasse Glue und die davon abgeleiteten Klassen HGlue und VGlue anhand von Deklarationsabschnitten genauer betrachten. Auf die Darstellung der Klasse Glue habe ich verzichtet, weil in Programmen in der Regel nur die abgeleiteten Klassen verwendet werden. Analog werden z.B. auch für Rule-Objekte, die in Kapitel 4 erklärt wurden, nur die abgeleiteten Klassen HRule und VRule benützt.

Objekte der Klasse HGlue realisieren Klebstoff in x-Richtung und werden in LRBoxes eingesetzt. Objekte der Klasse VGlue dienen als Klebstoff in y-Richtung und werden in TBBoxes verwendet. Die Deklaration der beiden Klassen zeigt die Syntax der beiden Konstruktoren:

```
class HGlue : public Glue
  {
    public:
      HGlue(Coord natural, Coord stretch,
            Coord shrink, float alignment = 0);
      ...
  };

class VGlue : public Glue
  {
    public:
      VGlue(Coord natural, Coord stretch,
            Coord shrink, float alignment = 0);
      ...
  };
```

Die Bedeutung der Kontruktorparameter ist in beiden Klassen analog. Die ersten drei Parameter regeln die natürliche Größe, die Dehnbarkeit (=Vergrößerbarkeit) und Stauchbarkeit(=Verkleinerbarkeit) des zu erzeugenden Glues. Der vierte Parameter dient zur Einstellung des Ursprungs des Glues. Für diesen Parameter sind Angaben zwischen 0.0 und 1.0 möglich. 0.0 setzt den Ursprung an die Stelle der kleinsten Koordinate des Glues. Entsprechend wird der Ursprung durch den Wert 1.0 zur größten Koordinate gesetzt, bei einem HGlue z.B. nach rechts. Welche Bedeutung die Lage des Ursprungs für die Layoutberechnung unter InterViews hat, erfahren Sie im Abschnitt 5.2.3.

Wichtig ist noch, daß InterViews die Attribute natürliche Größe, Dehnbarkeit und Stauchbarkeit von zusammengesetzten Glyphs, z.B. LRBoxen und TBBoxen, aus den entsprechenden Werten der Komponenten errechnet.

5.2.2 MonoGlyphs

MonoGlyphs enthalten genau einen Glyph, der als "Body" bezeichnet wird, und verändern dessen Aussehen oder Verhalten.

MonoGlyph:

Die Klasse MonoGlyph ist von Glyph abgeleitet. Sie ist abstrakt und bildet die Basisklasse für sämtliche "MonoGlyphs". MonoGlyph stellt Memberfunktionen zur Verfügung, die es erlauben, den Body abzufragen oder neu zu setzen. Die Semantik der Funktionen wird aus der Klassendeklaration deutlich:

```
class MonoGlyph : public Glyph
  {
    protected:
      MonoGlyph();

    public:
      virtual void body(Glyph*);
      virtual Glyph* body() const;
      ...
  };
```

Background:

Ein Background-Objekt hinterlegt seinen Body mit einer bestimmten Farbe. Ohne einen solchen Hintergrund enthalten Glyphs, die ihren Hintergrund nicht selbst zeichnen (z.B. Labels), also "durchsichtig" sind, Teile des ursprünglichen Fensterhintergrunds. Dieser besteht je nach Windowmanager häufig aus zufälligen Speichermustern und kann sehr störend sein. Die Klasse Background ist folgendermaßen deklariert:

```
class Background
  {
    public:
      Background(Glyph*, const Color*);
      ...
  };
```

Border:

Ein Objekt dieser Klasse zeichnet einen Rand vorgegebener Breite um seinen Body. Dabei ist zu beachten, daß der Rand noch innerhalb des Bodys gezeichnet wird. Ein Border-Objekt fügt also keinen zusätzlichen Platz für den Rand an den Body an. Die Deklaration ist selbsterklärend:

```
class Border : public MonoGlyph
  {
    public:
      Border(Glyph*, const Color*, Coord thickness);
      ...
  };
```

Margin:

Ein Margin reserviert zusätzlichen Platz rund um einen Glyph, zeichnet aber keinen Rand. Um einen Rand außerhalb eines Glyphs zu erzeugen, wird der zu umrandende Glyph in ein Margin-Objekt eingefügt. Das entstehende Objekt wird wiederum als Body einem Border-Objekt übergeben. Dabei sollte natürlich die Breite des Borders dem zusätzlichen Platz entsprechen, der durch das Margin-Objekt bereitgestellt wird. Anhand eines Deklarationsausschnitts sehen wir uns einen der verfügbaren Margin-Konstruktoren an:

```
class Margin : public MonoGlyph
  {
    public:
      Margin(Glyph*, Coord left, Coord right,
                     Coord bottom, Coord top);
      ...
  };
```

Die vier Coordparameter des hier vorgestellten Konstruktors beschreiben den zusätzlichen Platz, der links, rechts, unterhalb und oberhalb des Bodys allokiert werden soll.

Center:

Wie bereits in Abschnitt 2.3 erwähnt wurde, haben alle Glyphs einen Ursprung. Ein Objekt der Klasse Center kann die Position des Ursprungs seines Bodys verändern. Vor der Beschreibung dieser Klasse sollen aber noch einige allgemeine Punkte angesprochen werden, die den Ursprung eines Glyphs betreffen.

Jedem Glyph wird in der Memberfunktion Glyph::allocate mittels der vom System übergebenen Allocation-Referenz mitgeteilt, welcher Koordinatenbereich auf dem Canvas zu seiner Verfügung steht. Durch Memberfunktionen der Klasse

Allocation kann dieser Bereich abgefragt werden. Außerdem kann von dem Allocation-Objekt der Ursprung des Glyphs abgefragt werden. Der Ursprung liegt innerhalb des allokierten Bereichs. Seine Position wird durch die beiden alignment-Parameter des Requisition-Objekts festgelegt, das in Glyph::request besetzt wird. Es gibt für jede Dimension einen solchen alignment-Parameter, der mit Werten zwischen 0.0 und 1.0 besetzt werden muß. 0.0 bedeutet, daß der Ursprung in der betreffenden Dimension bei der kleinsten Koordinate liegt. Entsprechend setzt 1.0 den Ursprung bei der größten Koordinate des allokierten Bereichs fest. Nachfolgende Beispiele sollen diesen Sachverhalt verdeutlichen:

x-Alignment	y-Alignment	Position des Ursprungs
0.0	0.0	links unten
0.5	0.0	unten mitte
1.0	0.0	unten rechts
0.0	0.5	links mitte
0.5	0.5	Zentrum
1.0	1.0	rechts oben

Das Aussehen von zusammengesetzten Glyphs wie z.B. LRBoxen und TBBoxen wird von der Lage der Ursprungskoordinaten der Boxkomponenten wesentlich beeinflußt. Die genauen Zusammenhänge zwischen Ursprungspositionen der Komponenten und Aussehen der entstehenden Box werden in Abschnitt 5.2.3 erklärt. Die Lage des Ursprungs von vordefinierten Glyphs (z.B. Labels) ist jedoch dem Programmierer nicht bekannt. Ebenso ist die Berechnung des Ursprungs eines zusammengesetzten Glyphs nicht zugänglich. Daher ist es in diesen Fällen oft sinnvoll, die Position des Ursprungs durch einen Center-MonoGlyph nachträglich zu ändern.

Um den Ursprung in x- bzw. y-Richtung zu verschieben, stellt InterViews die beiden Klassen HCenter und VCenter bereit:

```
class HCenter
  {
    public:
      HCenter(Glyph*, float align = 0.5);
      ...
  };

class VCenter
  {
    public:
      VCenter(Glyph*, float align = 0.5);
      ...
  };
```

HCenter-Objekte beeinflußen die horizontale Lage des Ursprungs, während VCenter-Objekte die vertikale Lage des Ursprungs festsetzen. Der Klassenname HCenter bzw. VCenter ist dabei etwas irreführend, weil nur ein align-Wert von 0.5 den Ursprung wirklich in die Mitte der jeweiligen Dimension setzt. Werte zwischen 0.0 und 0.5 setzen den Ursprung weiter nach links (HCenter) bzw. weiter nach unten (VCenter). Werte zwischen 0.5 und 1.0 verschieben den Ursprung nach rechts (HCenter) bzw. nach oben (VCenter).

5.2.3 Zusammengesetzte Glyphs

InterViews bietet mehrere Möglichkeiten an, Glyphs zusammenzusetzen. Wir sehen uns nur die beiden wichtigsten an, nämlich die beiden Klassen LRBox und TBBox. Dazu soll folgender Deklarationsausschnitt diskutiert werden:

```
class LRBox : public Box
  {
    public:
      LRBox(Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil);
      LRBox(unsigned int initial_size);
      virtual void append(Glyph*);
      ...
  };

class TBBox : public Box
  {
    public:
      TBBox(Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil, Glyph* = nil, Glyph* = nil,
            Glyph* = nil);
      TBBox(unsigned int initial_size);
      virtual void append(Glyph*);
      ...
  };
```

Den Standardkonstruktoren der beiden Klassen können bis zu zehn Komponenten übergeben werden. Weitere Komponenten können mit append eingefügt werden. Soll die LRBox bzw. TBBox mehr als zehn Komponenten erhalten, so ist es sinnvoller, den anderen Konstruktor mit der ungefähren Komponentenanzahl aufzurufen und die Komponenten einzeln mit append einzufügen. Eine Box alokiert nämlich zunächst Speicherplatz für zehn Komponenten. Werden mehr eingefügt, so muß InterViews bei jedem Einfügen die gesamte Liste neu allokieren. Bei Verwendung des zweiten Konstruktors kann von vornherein genügend Speicherplatz reserviert werden.

Im weiteren werden nur LRBoxes ausführlicher besprochen. Die Ergebnisse sind aber völlig analog auf TBBoxes übertragbar.

LRBoxes (left-right-boxes) ordnen ihre Komponenten von links nach rechts an. Dabei wird versucht, die unterschiedlichen Höhen der Komponenten unter Ausnutzung ihrer Dehnbarkeit und Stauchbarkeit auszugleichen. Meist können die Komponentenglyphs jedoch nicht ausreichend gedehnt oder gestaucht werden. Dazu gehören beispielsweise viele vordefinierte Glyphs wie die Objekte der Klasse Label. Deshalb wird beim Zusammensetzen von Glyphs zahlreich von HGlues und VGlues Gebrauch gemacht, die zur Auffüllung der Zwischenräume dienen. Diese Glues werden evtl. gedehnt oder gestaucht, bis sämtliche Zwischenräume aufgefüllt sind. Dehnungen und Stauchungen werden dabei mit Hilfe eines raffinierten Algorithmus durchgeführt. Innerhalb einer LRBox werden evtl. vorhandene Zwischenräume durch HGlues aufgefüllt. Bei der Berechnung des Layouts werden die Komponenten mit der größten Dehnbarkeit (in der Regel ist dies ein HGlue-Objekt) auch am stärksten gestreckt; entsprechendes gilt für die Stauchung.

Das Aussehen einer LRBox wird aber auch von den Ursprungskoordinaten seiner Komponenten beeinflußt. Die Komponenten werden so angeordnet, daß ihre Ursprünge auf einer gedachten waagrechten Linie liegen. Demzufolge sind nur die y-Koordinaten der Komponentenursprünge für das Aussehen der entstehenden LRBox entscheidend, während die x-Koordinaten keine Rolle spielen. Das Prinzip wird in Abbildung 5.4 verdeutlicht. Als Beispiel dient eine LRBox, die aus zwei Komponenten fester Größe besteht, d.h. die Komponenten können nicht gedehnt oder gestaucht werden. Auf Glues wurde hier verzichtet, weil nur das Prinzip veranschaulicht werden soll. In einer "korrekten" LRBox würde beispielsweise die niedrigere der beiden Komponenten zunächst zusammen mit einem oder zwei VGlues in einer TBBox zusammengefaßt und danach diese TBBox mit der anderen Komponente zu einer LRBox komponiert.

In Abbildung 5.4 sind für jede Komponente die entsprechenden Tupel (x-Alignment/y-Alignment) abgedruckt. Ist eines der Tupelkomponenten durch "–" ersetzt, so wird damit angezeigt, daß dessen Wert ohne Einfluß auf das von InterViews errechnete Layout bleibt. Die gestrichelte Linie zeigt die y-Position der Ursprünge an.

Das Boxlayout führt bei Anfängern wie bei fortgeschrittenen InterViews-Programmierern immer wieder zu Verwirrungen. Das InterViews-Referenzmanual [LIN 91] gibt zu diesem Thema nur unzureichende Informationen. Da das Boxprinzip von InterViews aber weitgehend aus TEX übernommen ist, können Sie aus Knuths TEXbook [KNU 86] wertvolle Hinweise entnehmen.

Als pragmatische Vorgehensweise ist folgendes zu empfehlen: Bauen Sie Ihre Glyphs mit LRBoxes und TBBoxes unter Zuhilfenahme von HGlues in LRBoxes

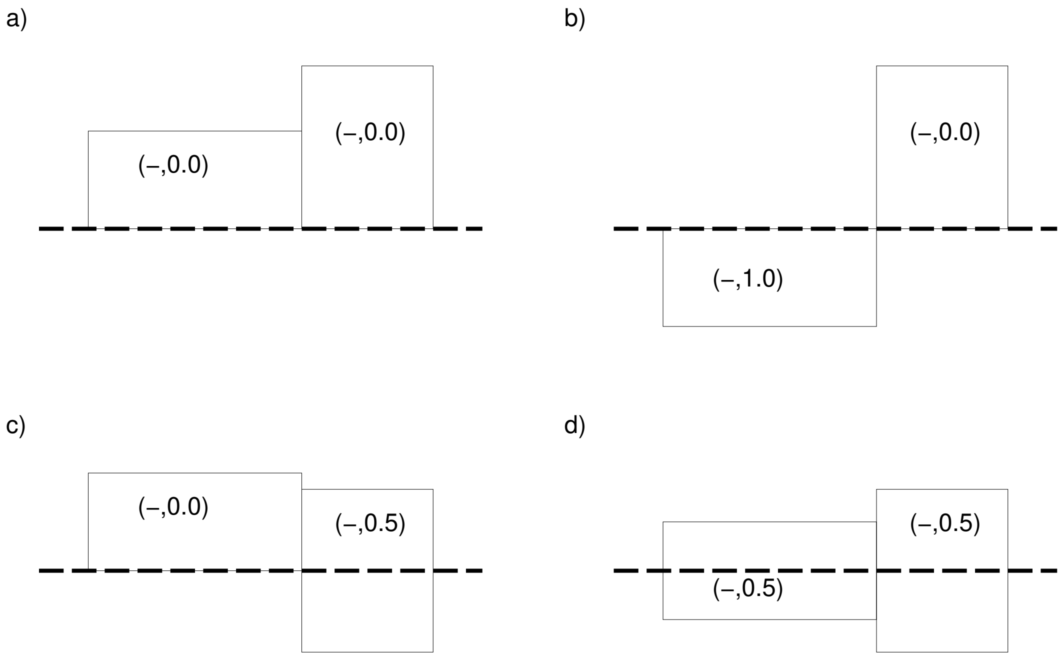

Abbildung 5.4: Einfluß der Ursprungskoordinaten der Komponenten auf das Aussehen einer LRBox

und VGlues in TBBoxes ohne Rücksicht auf die Ursprungspositionen auf. Erhalten Sie dabei nicht das gewünschte Ergebnis, müssen Sie mit HCenter- und VCenter-Objekten solange experimentieren, bis Sie Ihr Ziel erreicht haben.

Zu TBBoxes (top-bottom-boxes) ist noch zu bemerken, daß Sie Ihre Komponenten so von oben nach unten anordnen, daß die x-Koordinaten der Komponentenursprünge gleich sind. Das hat zur Folge, daß die Ursprünge der Boxkomponenten auf einer gedachten senkrechten Linie liegen.

5.3 Windows

Bereits in Abschnitt 1.3 wurden die Klassen für die Erzeugung von Bildschirmfenstern kurz erwähnt. Bisher wurde lediglich ein Fenster für jede Applikation geöffnet. Größere InterViews-Anwendungen machen aber häufig von mehreren Fenstern Gebrauch. Auch die Dialogbox, die in diesem Kapitel aufgebaut werden soll, wird in einem eigenen Fenster am Bildschirm dargestellt. Daher wollen wir uns an dieser Stelle eingehender mit dem Thema beschäftigen. Abbildung 5.5 zeigt die Ableitungsstruktur der Klassen für die Erzeugung von Bildschirmfenstern.

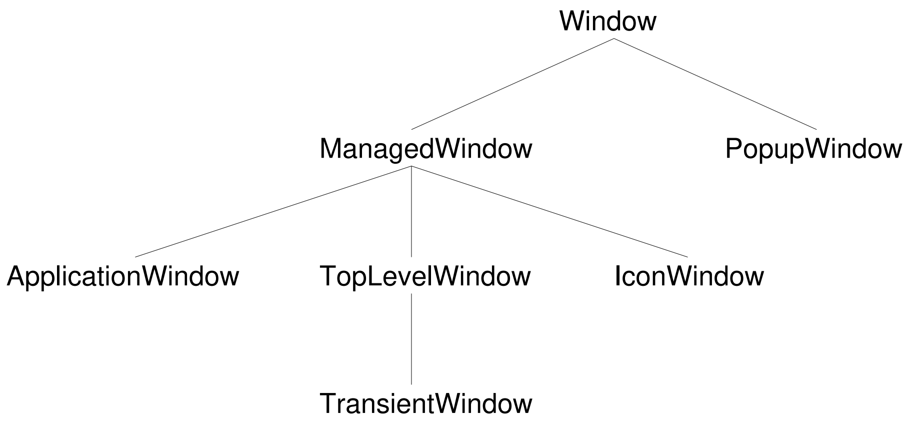

Abbildung 5.5: Ableitungsstruktur der Window-Klassen

Window:

Window ist die abstrakte Basisklasse aller dieser Klassen. Der für uns wichtige Teil der Klassendeklaration hat folgende Gestalt:

```
class Window
  {
    protected:
      Window(Glyph* = nil);

    public:
      virtual void place(Coord x, Coord y);
      virtual void align(float x, float y);
      virtual void map();
      virtual void unmap();
      ...
  };
```

Die Memberfunktion place plaziert das Fenster so am Bildschirm, daß sein Ursprung mit den angegebenen Koordinaten zusammenfällt. Die Lage des Fensterursprungs wird durch align festgelegt. Als Parameter erhält diese Routine zwei float-Werte zwischen 0.0 und 1.0. Die Semantik dieser Parameter folgt denselben Regeln, wie sie für die Festlegung eines Glyphursprungs bereits ausführlich beschrieben wurden. Beispielsweise setzt das Argumentepaar (0.5,0.5) den Fensterursprung in die Mitte des Fensters. Window::map bildet das Fenster auf dem Bildschirm ab, während es mittels unmap wieder vom Bildschirm entfernt wird. Wichtig ist, daß die Objekte aller von Window abgeleiteter Klassen keine shared objects sind und mit delete gelöscht werden müssen.

Von der Klasse Window sind gemäß Abbildung 5.5 zwei Klassen abgeleitet, die im folgenden beschrieben werden.

PopupWindow:

Objekte dieser Klasse werden für Fenster verwendet, die nur eine sehr kurze Lebensdauer haben. Außerdem werden solche Fenster ohne Kenntnis des Windowmanagers abgebildet, d.h. sie erhalten keinen Rahmen und können auch nicht während des Programmlaufs vergrößert oder verkleinert werden. PopupWindows werden bei der Konstruktion von Pulldown- oder Popupmenüs verwendet. Da uns jedoch die Klasse Kit diese Konstruktionsarbeit weitgehend abnimmt, wird hier auf eine weitergehende Darstellung der Klasse PopupWindow verzichtet.

ManagedWindow:

Die Klasse ManagedWindow bildet die abstrakte Basisklasse für alle Klassen, die Fenster für die Interaktion mit einem Windowmanager erzeugen. So dienen auch ihre Memberfunktionen zum größten Teil der Erledigung von Verwaltungsaufgaben. Einen kleinen Ausschnitt aus der großen Anzahl der verfügbaren Routinen zeigt der hier abgedruckte Teil der Klassendeklaration:

```
class ManagedWindow : public Window
  {
    protected:
      ManagedWindow(Glyph*);

    public:
      virtual void name(const char*);
      virtual void icon_name(const char*);
      virtual void iconify();
      virtual void deiconify();
      ...
  };
```

Mit Hilfe der Memberfunktion name kann einem Fenster ein neuer Titel gegeben werden. Der Fenstertitel erscheint in der Kopfzeile des zugehörigen Bildschirmfensters. Defaultwert ist der Name des Programms, der das Fenster erzeugt hat. Sicherlich ist Ihnen bekannt, daß ein Fenster durch Anklicken eines bestimmten Feldes im Fensterrahmen als Icon dargestellt werden kann. Durch Anklicken dieses Icons wird das Fenster wieder in Normalgröße auf dem Bildschirm abgebildet. Dieselbe Funktionalität bieten die beiden Routinen iconify und deiconify. Mittels icon_name kann der Name des Icons festgelegt werden. Darüberhinaus sind unter anderen noch Memberfunktionen vorgesehen, durch die das Aussehen der Icons beeinflußt werden kann. Diese und weitere Routinen werden im Referenzmanual [LIN 91] beschrieben.

ApplicationWindow:

Das Hauptfenster einer InterViews-Applikation sollte von einem Objekt der Klasse ApplicationWindow erzeugt werden. Außerdem darf jede Anwendung nur ein Hauptfenster haben. Die Klasse ApplicationWindow stellt außer dem Konstruktor keine zusätzlichen Memberfunktionen bereit. Jedoch sind sämtliche von Window bzw. ManagedWindow ererbten Memberroutinen verfügbar. Die Deklaration sieht wie folgt aus:

```
class ApplicationWindow : public ManagedWindow
  {
    public:
      ApplicationWindow(Glyph*);
      ...
  };
```

Der Konstruktor erhält als Parameter einen Zeiger auf die Wurzel einer Glyphhierarchie, die mindestens ein Element enthält. Diese Hierarchie wird bei der Abbildung des Fensters automatisch als Fensterinhalt am Bildschirm dargestellt.

TopLevelWindow:

Alle anderen "normalen" Fenster einer Applikation sind vom Typ TopLevelWindow. Die Deklaration der Klasse TopLevelWindow hat folgende Gestalt:

```
class TopLevelWindow : public ManagedWindow
  {
    public:
      TopLevelWindow(Glyph*);
      virtual void group_leader(ManagedWindow*);
      virtual ManagedWindow* group_leader() const;
      ...
  };
```

Der Konstruktor erhält als Argument wiederum einen Zeiger auf die Wurzel einer Glyphhierarchie. Als Besonderheit besitzen TopLevelWindows einen sogenannten "group_leader". Diese Rolle übernimmt im Regelfall das Hauptfenster der Anwendung, aber abhängig vom verwendeten Windowmanager können auch weitere Gruppierungen sinnvoll sein, was jedoch hier nicht weiter ausgeführt wird. Der "group_leader" kann durch die beiden group_leader-Memberfunktionen gesetzt bzw. zurückgeliefert werden. Manche Windowmanager erlauben es, bestimmte Operationen (z.B. Ikonifizieren), die auf dem "group_leader" ausgeführt werden, zugleich auf alle Fenster der Gruppe anzuwenden.

TransientWindow:

Objekte dieser von TopLevelWindow abgeleiteten Klasse werden für temporäre Fenster wie z.B. Dialogboxen verwendet. Im Gegensatz zu PopupWindows werden TransientWindows aber vom Windowmanager verwaltet. Allerdings werden sie von manchen Windowmanagern zur Unterscheidung von "normalen" Fenstern beispielsweise mit einem anderen Rahmen oder Icon gekennzeichnet. Der für uns interessante Deklarationsausschnitt sieht so aus:

```
class TransientWindow : public TopLevelWindow
  {
    public:
      TransientWindow(Glyph*);
      virtual void transient_for(ManagedWindow*);
      virtual ManagedWindow* transient_for() const;
      ...
  };
```

Jedes TransientWindow sollte mit dem Fenster assoziiert werden, für das es temporäre Aufgaben, z.B. die Anzeige einer Dialogbox, ausführt. Manche Windowmanager entfernen ein TransientWindow beispielsweise automatisch vom Bildschirm, wenn das assoziierte Fenster ikonifiziert wird. Mittels der beiden transient_for-Memberfunktionen wir das assoziierte Fenster gesetzt bzw. abgefragt.

IconWindow:

Objekte dieser Klasse werden am Bildschirm abgebildet, wenn ihr assoziiertes Fenster ikonifiziert wird, d.h. sie ersetzen bzw. ergänzen die herkömmlichen Icons. Wir verwenden diese Klasse nicht, sie wird hier nur der Vollständigkeit halber erwähnt.

5.4 Weitere notwendige InterViews-Klassen

In diesem Abschnitt werden weitere Memberfunktionen der Klasse Kit beschrieben. Außerdem werden die Klassen Bevel, Button und FieldEditor neu eingeführt.

Kit:

Der folgende Ausschnitt zeigt die Deklaration weiterer Memberfunktionen der Klasse Kit:

```
class Kit
  {
    public:
      ...
      virtual Glyph* inset_frame(Glyph*,
                                 Style*) const;
      virtual Glyph* outset_frame(Glyph*,
                                  Style*) const;
      virtual Glyph* label(const char*,
                           Style*) const;
      Button* simple_push_button(const char*,
                                 Style*, Action*);
      ...
  };
```

Die Routine inset_frame erzeugt um den als Zeiger übergebenen Glyph einen Rahmen (Objekt der Klasse Bevel, siehe unten) derart, daß der Inhalt des Rahmens (der Glyph) gegenüber seiner Umgebung leicht zurückgesetzt erscheint. Analog konstruiert outset_frame einen Rahmen, dessen Inhalt leicht hervorgehoben erscheint. Die Memberfunktion label erzeugt aus dem Argumentstring ein Label-Objekt. Die Routine simple_push_button konstruiert ein Button-Objekt (die Klasse Button wird gleich im Anschluß erklärt), das den übergebenen String enthält. Außerdem wird noch ein Zeiger auf ein Action-Objekt als Parameter angegeben. Die zugehörige Aktion wird ausgeführt, wenn auf dem Bildschirm dieser Button angeklickt wird.

Die eben vorgestellten Routinen erhalten als zweites Argument einen Zeiger auf ein Style-Objekt, in dem z.B. die Farben für die Erzeugung der Rahmen festgelegt sind. Der plastische Effekt, den die Rahmen bewirken, wird nämlich lediglich durch eine geschickte Färbung der Rahmenseiten bewirkt.

Bevel:

Die Klasse Bevel ist von MonoGlyph abgeleitet. Objekte dieser Klasse versehen ihren Body mit einem Rahmen, der sie vor- oder zurückgesetzt erscheinen lßt. Die beidenKit-Memberfunktionen inset_frame und outset_frame erzeugen solche Bevel-Objekte. Die Klasse Bevel wird hier nur der Vollständigkeit halber erwähnt. Die Memberfunktionen der Klasse Kit ersparen es dem InterViews-Programmierer, sich direkt mit Bevels auseinanderzusetzen.

Button:

Objekte dieser Klasse gehören zu den meistverwendeten Bausteinen graphischer Oberflächen. Buttons werden z.B. eingesetzt, wenn in einer Dialogbox eine bestimmte Eingabe bestätigt ("OK-Button") oder zurückgewiesen ("Cancel-Button") werden soll. Um eine möglichst einheitliche Benutzeroberfläche zu erhalten, sollten Buttons stets mit Hilfe eines Kit-Objekts erzeugt werden. Aus diesem Grund wird die Klasse Button hier nicht näher vorgestellt. Wichtig ist nur, daß sie von Glyph abgeleitet ist. Damit können Buttons durch Einbau in eine Glyphhierarchie in eine Benutzeroberfläche eingefügt werden.

FieldEditor:

In einer Dialogbox werden in der Regel bestimmte Eingaben mit Hilfe von kleinen Editoren erledigt. Auch für diesen Zweck stellt InterViews Klassen zur Verfügung, z.B. die Klasse FieldEditor. Objekte dieser Klasse realisieren Editoren für eine Zeile Text. Leider ist die Kit-Klasse in der gegenwärtigen InterViews-Version noch nicht vollständig. So liegt beispielsweise für die Konstruktion eines FieldEditors noch keine Kit-Memberfunktion vor. Deshalb ist die einfache Erzeugung eines FieldEditors mit einem bestimmten Erscheinungsbild derzeit nicht möglich. Daher muß der Programmierer mittels des Konstruktors selbst einen FieldEditor erzeugen und versuchen, ihm mit Hilfe von outset_frames oder inset_frames das gewünschte Erscheinungsbild zu geben.

Anhand eines Deklarationsausschnitts soll die Klasse FieldEditor nun diskutiert werden:

```
class FieldEditor : public MonoGlyph
  {
    public:
      FieldEditor(const char *sample, Style*,
                  Action *ok = nil, Action *cancel = nil);
      const char* text() const;
      ...
  };
```

FieldEditor ist von MonoGlyph und somit auch von Glyph abgeleitet. Daher kann ein Objekt dieser Klasse in eine Glyphhierarchie und damit in eine Benutzeroberfläche eingefügt werden.

Bevor wir uns den Memberfunktionen zuwenden, soll zunächst die Bedienung eines FieldEditors erklärt werden. Folgende wichtige Tastenkombinationen steuern den Editiervorgang:

```
<ctrl>-b:     ein Zeichen nach links (backspace)
<ctrl>-f:     ein Zeichen nach rechts (forward)
<ctrl>-a:     an den Anfang der Zeile springen
<ctrl>-e:     an das Ende der Zeile springen
<ctrl>-d:     das Zeichen unter dem Cursor loeschen
              (delete)
<ctrl>-h:     das Zeichen links vom Cursor loeschen
```

Durch Drücken der Returntaste wird der Editiervorgang regulär beendet. Wird ein Escapezeichen eingegeben, so wird der Editiervorgang abgebrochen.

Aus den verfügbaren Memberfunktionen der Klasse FieldEditor habe ich die beiden wichtigsten ausgewählt. Der Konstruktor erhält bis zu vier Parameter. Das erste Argument wird mit einem Beispielstring besetzt, der für die Berechnung der Größe des Editors verwendet wird und bei der Abbildung des Editors am Bildschirm im Editierfenster erscheint. Das als Zeiger übergebene Style-Objekt regelt das Aussehen des Editors. Das optional anzugebende Action-Objekt ok wird aktiviert, wenn der Editiervorgang regulär beendet wird. Wird dieser Vorgang dagegen abgebrochen, so wird das Action-Objekt ausgeführt, auf das der Zeiger cancel zeigt. In beiden Fällen bedeutet die Angabe nil, daß in der entsprechenden Situation keine Aktion zur Ausführung gelangt.

5.5 Implementierung der Aufgabenstellung

Nach diesen umfangreichen Erklärungen, in denen Sie einiges über die InterViews-Konzepte gelernt haben, wird nun die Implementierung der Aufgabenstellung aus Abschnitt 5.1 vorgestellt. Wenn Sie dieses Programm verstehen, sind Sie bereits in der Lage, größere InterViews-Anwendungen selbst zu schreiben.

Der Sourcecode ist auf die Files header.h, Linie.h, Linie.c, TestMenu.h, TestMenu.c, TestDialog.h, TestDialog.c und dialogtest.c verteilt. Die Datei header.h wurde gegenüber Kapitel 4 um die InterViews-Includedateien für Glues, FieldEditors und Buttons erweitert. Außerdem wurde das File TestDialog.h inkludiert, das die Deklaration unserer neuen Klasse TestDialog (für die Erzeugung der Dialogbox) enthält:

```
#ifndef header_h
#define header_h

#include <InterViews/session.h>
#include <InterViews/display.h>
#include <InterViews/style.h>
#include <InterViews/window.h>
#include <InterViews/glyph.h>
#include <InterViews/background.h>
#include <InterViews/rule.h>
#include <InterViews/box.h>
#include <InterViews/glue.h>
#include <InterViews/color.h>
#include <InterViews/brush.h>
#include <InterViews/event.h>
#include <InterViews/handler.h>
#include <InterViews/hit.h>
```

```
#include <InterViews/action.h>
#include <IV-look/kit.h>
#include <IV-look/menu.h>
#include <IV-look/field.h>
#include <IV-look/button.h>

#include <iostream.h>
#include <strings.h>

#include "Linie.h"
#include "TestDialog.h"
#include "TestMenu.h"

#endif
```

Die beiden Dateien Linie.h und Linie.c sind gegenüber Kapitel 3 und 4 unverändert und können übernommen werden. Dagegen haben TestMenu.h und TestMenu.c im Vergleich zu Kapitel 4 kleine Änderungen erfahren, die im wesentlichen mit der Erzeugung und Löschung der Dialogbox zusammenhängen.

TestMenu.h hat nun folgende Gestalt:

```
#ifndef TestMenu_h
#define TestMenu_h

class TestMenu : public TBBox
{
  public:
    TestMenu();
    Menu* CreateMenu();
    void set_window(ApplicationWindow*);
    void copyright();
    void preferences();
    void call_dialogbox();
    void delete_dialogbox();
  private:
    ApplicationWindow *menuwindow;
    TransientWindow *dialogwindow;
    TestDialog *dialog;
    FieldEditor *editor;
};

declare(ActionCallback,TestMenu)

#endif
```

Die neue Memberfunktion set_window wird mit einem Zeiger auf ein ApplicationWindow aufgerufen. Diese Routine wird von unserem Hauptprogramm (aus dialogtest.c) mit einem Zeiger auf das Hauptfenster aufgerufen, das wie bereits in Kapitel 4 das Pulldownmenü enthält. Dieser Zeiger wird in set_window in der Membervariablen menuwindow abgespeichert.

Die Memberfunktionen call_dialogbox und delete_dialogbox sind für das Erzeugen und Löschen der Dialogbox zuständig. Neben dem oben erwähnten menuwindow werden noch drei weitere private Membervariablen eingeführt. Die Variable dialogwindow wird mit einem Zeiger auf das Fenster belegt, das die Dialogbox enthält. Gemäß den in Abschnitt 5.3 angegebenen Kriterien wurde TransientWindow als geeignete Klasse für dieses Fenster gewählt. Die Variable dialog wird einen Zeiger auf die erzeugte Dialogbox erhalten. Ein Zeiger auf den FieldEditor, der in die Dialogbox eingebaut wird, wird in editor abgespeichert.

Das gegenüber Kapitel 4 ebenfalls veränderte File TestMenu.c hat nun folgendes Aussehen:

```
#include "header.h"

implement(ActionCallback,TestMenu)


TestMenu::TestMenu() : TBBox()
{
  menuwindow = nil;
  dialogwindow = nil;
  dialog = nil;

  Session *session = Session::instance();

  append(CreateMenu());
  append(new HRule(session->style()->foreground(),1.0));
  append(new Linie());
}


Menu* TestMenu::CreateMenu()
{
  Session *session = Session::instance();
  Kit *kit = Kit::instance();
  Style *style = session->style();

  Menu *mb = kit->menubar(style);
  Menu *file = kit->pulldown(style);
```

```
  Menu *options = kit->pulldown(style);

  Action *copyright = new ActionCallback(TestMenu)
                        (this,&TestMenu::copyright);

  Action *preferences = new ActionCallback(TestMenu)
                          (this,&TestMenu::preferences);

  Action *dialogbox = new ActionCallback(TestMenu)
                        (this,&TestMenu::call_dialogbox);

  file->add_item(kit->simple_menu_item("Copyright",style),
                 copyright);

  file->add_item(kit->simple_menu_item("Quit",style),
                 kit->quit());

  options->add_item(kit->simple_menu_item("Preferences",
                                          style),
                    preferences);

  options->add_item(kit->simple_menu_item("DialogBox",style),
                    dialogbox);

  mb->add_item(kit->simple_menubar_item("File",style),
               file);
  mb->add_item(kit->simple_menubar_item("Options",style),
               options);

  return mb;
}


void TestMenu::set_window(ApplicationWindow *aw)
{
  menuwindow = aw;
}


void TestMenu::copyright()
{
  cerr << "TestMenu::copyright called\n";
}
```

```
void TestMenu::preferences()
{
  cerr << "TestMenu::preferences called\n";
}


void TestMenu::call_dialogbox()
{
  if ( dialogwindow != nil )
    {
      cerr << "DialogBox already called\n";
      return;
    }

  Session *session = Session::instance();
  Display *display = session->default_display();
  Style *style = session->style();
  Kit *kit = Kit::instance();

  Action *pushaction = new ActionCallback(TestMenu)
                         (this,&TestMenu::delete_dialogbox);

  editor = new FieldEditor("Hans Mustermann",style);
  dialog = new TestDialog(editor,pushaction);

  dialogwindow = new TransientWindow(kit->outset_frame(dialog,
                                                        style));

  dialogwindow->align(0.5,0.5);
  dialogwindow->place(display->width()/2,display->height()/2);
  dialogwindow->name("DialogBox");
  dialogwindow->group_leader(menuwindow);
  dialogwindow->transient_for(menuwindow);
  dialogwindow->map();
}


void TestMenu::delete_dialogbox()
{
  cerr << "You entered: " << editor->text() << "\n";
```

```
  dialogwindow->unmap();
  delete dialogwindow;

  dialogwindow = nil;
}
```

Der Konstruktor besetzt nun zusätzlich die neueingeführten Membervariablen menuwindow, dialogwindow und dialog mit nil. In CreateMenu ist die Zeile neu, in der die Aktion "dialogbox" erzeugt wird. Die Ausführung dieser Aktion führt zum Aufruf der Routine call_dialogbox. Außerdem wird mittels options–>add_item der Eintrag "DialogBox" zusammen mit der zugehörigen Aktion "dialogbox" in das Pulldownmenü options eingefügt.

In der neuen Memberfunktion set_window wird der übergebene Zeiger auf ein ApplicationWindow in der Membervariablen menuwindow abgespeichert. Dabei wird erwartet, daß diese Routine mit einem Zeiger auf das Hauptfenster der Applikation aufgerufen wird, das auch das Pulldownmenü enthält. Der Zeiger auf dieses Fenster wird benötigt, um das Dialogboxfenster mit dem Hauptfenster assoziieren zu können.

Die beiden Routinen copyright und preferences bleiben unverändert. Wichtig ist aber die neue Routine call_dialogbox. Sie wird automatisch aufgerufen, wenn der Menüeintrag "DialogBox" aktiviert wird. Dies wird von der Aktion "dialogbox" bewirkt.

In call_dialogbox wird zunächst überprüft, ob der Zeiger auf das Dialogfenster nil ist. Falls dies nicht gilt, wurde inzwischen schon einmal call_dialogbox aufgerufen und dabei eine Dialogbox mit zugehörigem Fenster erzeugt, die aber noch nicht gelöscht wurde und sich daher noch auf dem Bildschirm befindet. In diesem Fall wird eine entsprechende Warnmeldung ausgegeben und die Ausführung der Routine beendet. Ansonsten werden nun die Dialogbox und das dazugehörige Fenster konstruiert. Dazu werden zunächst Zeiger auf das Session-Objekt, das Defaultdisplay, das Kit-Objekt und den Defaultstyle gesetzt. Danach wird die Aktion "pushaction" erzeugt, die bei der Aktivierung des OK-Buttons der Dialogbox ausgeführt werden soll. Diese Aktion bewirkt den Aufruf der Memberroutine delete_dialogbox. Schließlich wird noch ein FieldEditor mit dem Samplestring "Hans Mustermann" und dem Defaultstyle erzeugt. Der Zeiger auf diesen Editor wird in der Membervariablen editor festgehalten. Mit den Parametern editor und pushaction wird nun mittels unserer eigenen Klasse TestDialog die eigentliche Dialogbox konstruiert. Der Zeiger auf dieses Objekt wird in der Membervariablen dialog festgehalten. Danach wird ein TransientWindow erzeugt, das die Dialogbox enthalten soll. Die Dialogbox wird aber nicht direkt in das Fenster eingefügt, sondern vorher noch mit Hilfe des Kit-Objekts in einen Bevel gepackt. Dadurch erhält auch die Dialogbox das optisch ansprechende plastische Erscheinungsbild. Durch dialogwindow–>align(0.5,0.5) wird der Fensterursprung genau in die Fenstermitte gelegt. In der folgenden Zeile wird das Fenster so plaziert, daß sein Ursprung mit der Bildschirmmitte zusammenfällt. Außerdem erhält das Fenster den Namen "DialogBox". In den nächsten beiden Zeilen wird das mittels set_window gesetzte menuwindow, das Hauptfenster der Applikation, benötigt. Durch

```
dialogwindow->group_leader(menuwindow);
```

wird es als "groupleader" der Fenstergruppe erklärt, die lediglich aus dem Hauptfenster und dem Dialogfenster besteht. Mittels

```
dialogwindow->transient_for(menuwindow);
```

wird darüberhinaus dem Windowmanager bekanntgegeben, daß das Dialogfenster eine temporäre Aufgabe für das Hauptfenster (= Menüfenster) erledigt. Durch den abschließenden Aufruf der Window-Memberfunktion map wird das Dialogfenster zusammen mit der darin enthaltenen Dialogbox auf dem Bildschirm abgebildet.

Wurde in der Dialogbox der OK-Button betätigt, so wird die Routine delete_dialogbox aktiviert (vgl. die Definition der Aktion pushaction). In delete_dialogbox wird zunächst der editierte Text auf der Standardfehlerausgabe ausgegeben. Dieser Text kann durch die Memberfunktion FieldEditor::text (vgl. 5.4) erhalten werden. Danach wird das Fenster durch unmap vom Bildschirm entfernt und das zugehörige Objekt gelöscht. Die Membervariable dialogwindow wird schließlich noch explizit auf nil gesetzt.

Sicher werden Sie sich jetzt fragen, weshalb weder der FieldEditor noch das TestDialog-Objekt selbst explizit gelöscht werden. Dies liegt daran, daß beide Klassen Glyph und somit Resource als Basisklasse haben. Damit gehören sie zu den shared objects, die ja niemals durch delete gelöscht werden dürfen. Sie wissen sicherlich noch, daß shared objects mit ref- und unref- Aufrufen verwaltet werden und nach entsprechend vielen unref-Aufrufen automatisch gelöscht werden. Wie sich dieser Mechanismus auf unser Beispiel auswirkt, wird in Abschnitt 5.6 nochmals ausführlich erklärt. Bis dahin sollten Sie mir glauben, daß alle Objekte, aus denen die Dialogbox aufgebaut ist, korrekt gelöscht werden.

Damit ist die Diskussion der Klasse TestMenu abgeschlossen. Im folgenden wenden wir uns der neuen Klasse TestDialog zu. Die Deklaration der Klasse ist in der Datei TestDialog.h zu finden:

```
#ifndef TestDialog_h
#define TestDialog_h

class TestDialog : public TBBox
{
  public:
    TestDialog(FieldEditor*,Action*);
};

#endif
```

Die Klasse TestDialog wird dazu verwendet, eine Dialogbox zu konstruieren. Die Dialogbox besteht aus einer Zusammensetzung von LRBoxes, TBBoxes, HGlues, VGlues und anderen Objekten (siehe Abbildung 5.6). Oberstes Element der Glyphhierarchie ist eine TBBox. Da wir uns weiterhin an die InterViews-Richtlinie halten wollen, für eigene Objekte eigene Klassen zu schaffen, leiten wir die Klasse TestDialog einfach von TBBox ab und haben damit das oberste Element der Glyphhierarchie, die die gesamte Dialogbox beschreibt, lediglich "umbenannt." Als einzig neue Memberfunktion wird der Konstruktor deklariert, der den Zusammenbau der Box leistet.

Die Implementation der Klasse TestDialog ist in TestDialog.c enthalten:

```
#include "header.h"


TestDialog::TestDialog(FieldEditor *editor,
                       Action *pushaction) : TBBox()
{
  Kit *kit = Kit::instance();
  Style *style = Session::instance()->style();

  append(new VGlue(25.0,40.0,10.0));
  append(new LRBox(
                   new HGlue(25.0,100.0,10.0),
                   kit->outset_frame(kit->label(
                                 "Please enter your name:",
                                 style),
                                     style),
                   new HGlue(25.0,100.0,10.0)
                  )
        );
  append(new VGlue(15.0,30.0,5.0));
  append(new LRBox(
```

```
                     new HGlue(25.0,100.0,10.0),
                     kit->outset_frame(editor,style),
                     new HGlue(25.0,100.0,10.0)
                    )
          );
  append(new VGlue(15.0,30.0,5.0));
  append(new LRBox(
                     new HGlue(25.0,100.0,10.0),
                     kit->simple_push_button("OK",style,
                                             pushaction),
                     new HGlue(25.0,100.0,10.0)
                    )
          );
  append(new VGlue(25.0,40.0,10.0));
}
```

Um die Implementierung des Konstruktors, der die Dialogbox zusammensetzt, besser verstehen zu können, sehen wir uns zunächst die graphische Darstellung der realisierten Glyphhierarchie in Abbildung 5.6 an. Aus dieser Abbildung wird

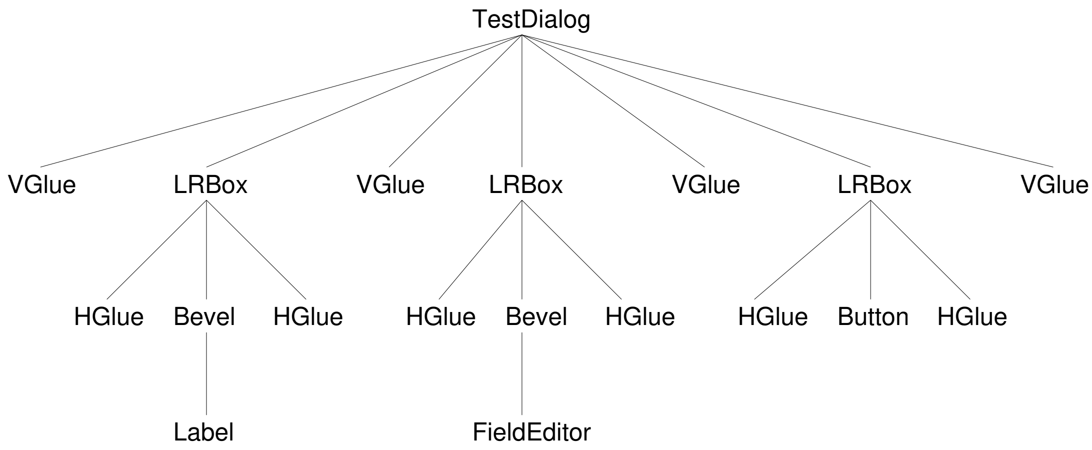

Abbildung 5.6: Struktur der Dialogbox

ersichtlich, daß die Dialogbox im wesentlichen aus einem Label, einem FieldEditor und einem Button besteht. Die übrigen Objekte sind für die Layoutgestaltung erforderlich. In Abbildung 5.7 ist die Struktur der erzeugten Dialogbox stilisiert dargestellt, wobei die einzelnen Komponenten zu Ihrer Orientierung beschriftet wurden. Die dargestellten Objektbegrenzungen werden selbstverständlich nicht auf dem Bildschirm abgebildet.

Da die äußere TBBox bereits beim Eintritt in den Konstruktor von TestDialog erzeugt wird, müssen ihre Komponenten nachträglich mittels append eingefügt

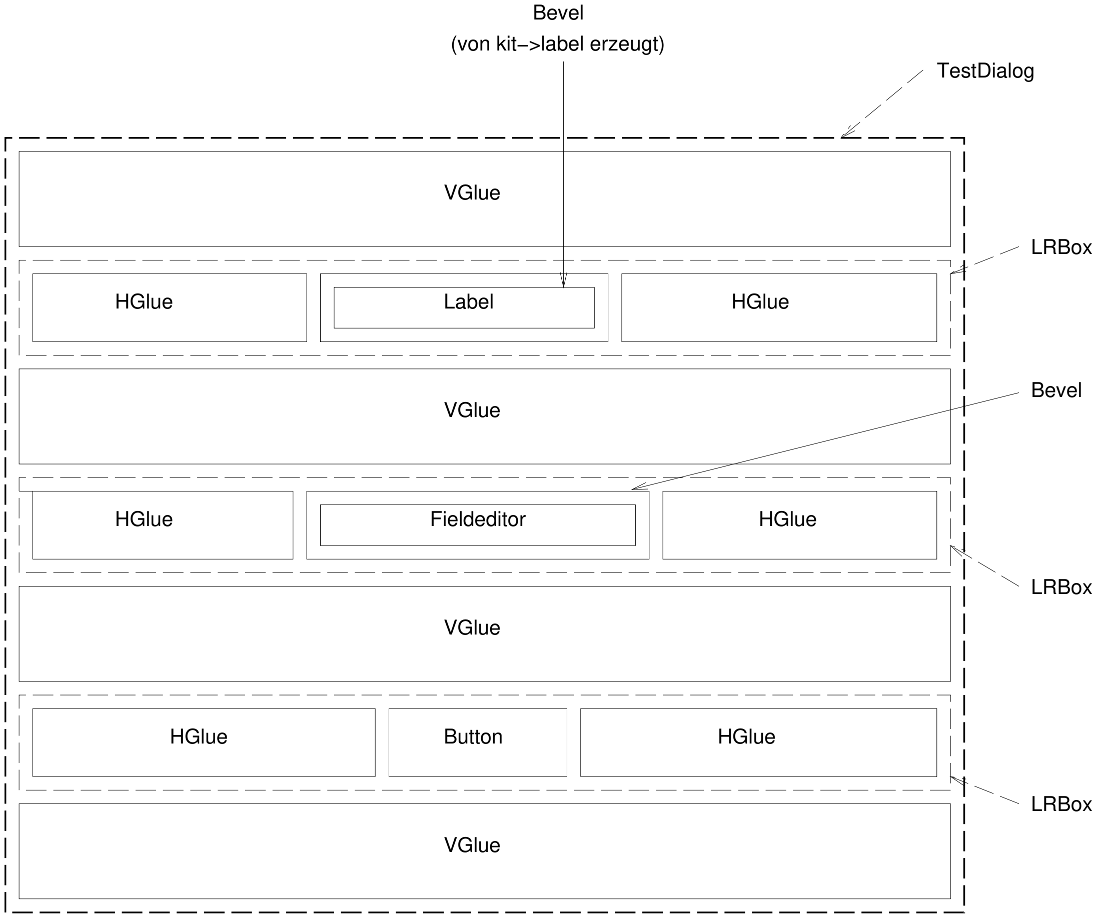

Abbildung 5.7: Stilisierte Darstellung der Dialogbox

werden. Die LRBoxen erhalten dagegen ihre Komponenten direkt als Konstruktorargumente. Diese Methode ist wegen der besseren Übersicht allgemein zu bevorzugen.

Beachten Sie, daß neben Objekten, die innerhalb einer Box zentriert werden sollen, Glues mit den gleichen charakteristischen Größen angegeben werden. Als Beispiel wird die LRBox betrachtet, die den Button enthält. Links und rechts des entsprechenden Button-Objekts befinden sich zwei HGlues mit denselben Parametern. Hätte der linke HGlue die gleiche natürliche Größe, jedoch eine kleinere Dehnbarkeit als der rechte HGlue, so würde in dem Fall, daß für die LRBox die Summe der natürlichen Größen der beiden HGlues nicht mehr ausreicht, der rechte HGlue stärker gedehnt. Damit würde sich der Button nach links verschieben.

Wenn Sie später dieses Programm ausprobieren, werden Sie feststellen, daß durch die Glyphhierarchie aus Abbildung 5.6 die gewünschte Dialogbox erzeugt wird. Damit kann in diesem Beispiel auf die Verwendung von HCenter- und VCenter-Objekten verzichtet werden.

Schließlich fehlt noch die Beschreibung des Hauptprogramms, das in dialogtest.c enthalten ist:

```
#include "header.h"


int main(int argc,char **argv)
{
  Session *session = new Session("DialogExample",argc,argv);
  Style *style = session->style();

  TestMenu *tm = new TestMenu();

  Background *bg = new Background(tm,style->background());

  ApplicationWindow *window = new ApplicationWindow(bg);

  tm->set_window(window);

  int status = session->run_window(window);

  window->unmap();
  delete window;

  return status;
}
```

Im Prinzip laufen hier die gleichen Schritte wie in menutest.c aus Kapitel 4 ab. Allerdings wird für den Zeiger auf das TestMenu-Objekt eine eigene Variable bereitgestellt, weil noch die Memberfunktion set_window aufgerufen werden muß. Außerdem wurde die Anweisung return–>run_window in vier Einzelanweisungen zerlegt. InterViews kehrt aus Session::run_window nach Aufruf von Session::quit zurück. Session::quit wird von der Aktion aufgerufen, die durch Kit::quit erzeugt und von dem Menüpunkt "Quit" aktiviert wurde. Zu dem Zeitpunkt, zu dem das System aus der Eingabeabfrageschleife zurückkehrt, befindet sich das ApplicationWindow aber noch auf dem Bildschirm. In den vorangegangenen Beispielen wurde an diesem Punkt die main-Routine sofort mittels return verlassen und damit der zugehörige UNIX-Prozeß beendet. Mit der Beendigung des Prozesses werden evtl. noch vorhandene Bildschirmfenster automatisch entfernt und durch die Freigabe des Prozeßadreßraums evtl. noch vorhandene Objekt gelöscht. Um diese Vorgänge zu verdeutlichen, wurden die einzelnen Schritte hier einmal explizit durchgeführt.

5.6 Löschen von InterViews-Objekten

Allgemein gilt folgender Grundsatz: *Durch new erzeugte Objekte einer Klasse, die von Resource abgeleitet ist (shared objects), dürfen nicht mit delete gelöscht werden. Sie werden durch Aufrufe der Memberfunktionen ref und unref verwaltet und gegebenenfalls automatisch zerstört. Mittels new erzeugte Objekte anderer Klassen werden mit delete gelöscht.*

Das Zusammenspiel dieser beiden Mechanismen soll nun anhand unseres Beispiels aus diesem Kapitel verdeutlicht werden. Dazu sehen wir uns die Funktion TestMenu::call_dialogbox an. Dort wird ein TransientWindow erzeugt, das einen Bevel (outset_frame) als Wurzel der im Fenster darzustellenden Glyphhierarchie erhält. Dieser Bevel enthält als Body die von TestDialog::TestDialog erzeugte Glyphhierarchie. Abbildung 5.8 faßt die Gesamtstruktur nochmals zusammen.

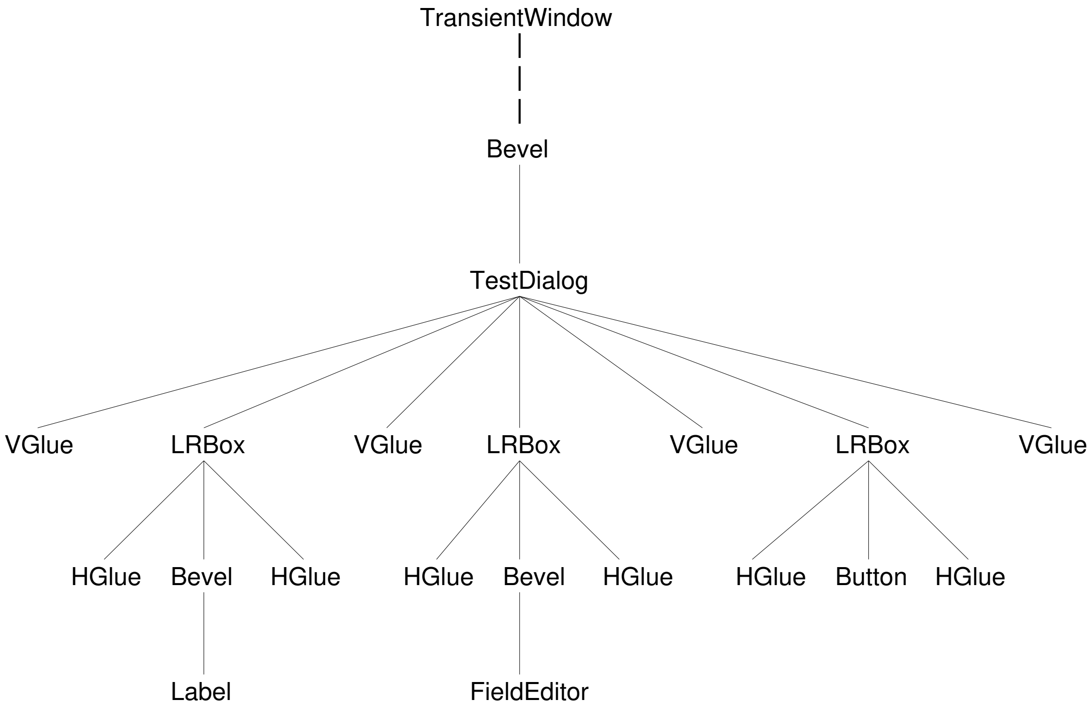

Abbildung 5.8: Struktur des Dialogfensters mit Dialogbox

Das TransientWindow erhält die Wurzel der Glyphhierarchie als Parameter, gehört aber selbst nicht dazu. Bevor diese Hierarchie in das Fenster eingefügt wird, wird sie in bottom-up Reihenfolge aufgebaut. Als Beispiel dient uns der Ast in Abbildung 5.8, der die Konstruktion der obersten LRBox aus Abbildung 5.7 beschreibt. Durch kit–>outset_frame wird ein Bevel-Objekt erzeugt, das als Body das Label-Objekt erhält. Die InterViews-Routinen sind so programmiert, daß das Label-Objekt beim Einfügen in den Bevel automatisch mittels ref referenziert wird. Der Konstruktor der danach erzeugten LRBox ruft für die beiden

HGlues und das Bevel-Objekt ref auf. Beim Einfügen der LRBox in das Objekt der von TBBox abgeleiteten Klasse TestDialog erledigt die Memberfunktion append den Aufruf von ref für diese LRBox. Das TestDialog-Objekt wird von dem Bevel-Objekt, das durch kit–>outset_frame konstruiert wird, mit ref referenziert. Der Konstruktor von TransientWindow ruft nun ref für das als Parameter übergebene Bevel-Objekt auf. Dadurch wurde schließlich für alle Komponenten der Glyphhierarchie einmal die Memberfunktion ref aufgerufen, d.h. alle Referenzzähler haben den Wert Eins.

Wird nun in TestMenu::delete_dialogbox

```
delete dialogwindow;
```

aufgerufen, so wird die Hierarchie top-down mittels unref vollautomatisch dereferenziert. Der Destruktor von TransientWindow, der durch delete aktiviert wurde, ruft unref für das oberste Bevel-Objekt auf. Dadurch erhält dessen Referenzzähler den Wert Null und das Objekt wird automatisch zerstört. Dabei wird der Body des Bevels, das TestDialog-Objekt, durch unref dereferenziert. Auch dessen Referenzzähler wird um eins auf null verringert. Folglich wird das TestDialog-Objekt gelöscht, wobei für sämtliche Komponenten unref aufgerufen wird. Die Referenzzähler der Komponenten werden dadurch auf null zurückgesetzt, was die Zerstörung dieser Objekte zur Folge hat. So pflanzt sich der Vorgang bis in die Blätter des Glyphgraphen fort. Sukzessive werden dabei alle Objekte durch unref-Aufrufe dereferenziert und automatisch gelöscht. Sie sollten sich noch einmal klarmachen, daß der InterViews-Programmierer den gesamten Löschvorgang durch

```
delete dialogwindow;
```

ausgelöst hat. Durch den ref-/unref-Mechanismus mit automatischer Löschung für shared objects realisiert InterViews eine Art "garbage collection".

5.7 Abschließende Bemerkungen

Auf gut 90 Seiten haben Sie nun die Grundlagen und vor allem in diesem Kapitel auch weiterführende Aspekte der InterViews-Programmierung kennengelernt. Dabei konnte nur ein Bruchteil der vordefinierten Klassen angesprochen werden. Aber auch die vorgestellten Klassen wurden häufig nur teilweise behandelt. Dennoch habe ich versucht, Ihnen die für den Einstieg in die InterViews-Programmierung nötigen Kenntnisse zu vermitteln. Sie sollten alle Beispiele in diesem Buch mit den angebotenen Informationen verstehen können. Eine gute Fertigkeit im Umgang mit dem überaus mächtigen Toolkit InterViews werden Sie aber erst erlangen, wenn Sie versuchen, eigene Applikationen zu schreiben. Dabei können Programme anderer InterViews-Programmierer immer wieder nützliche Anregungen bieten. Außerdem haben Sie jetzt genügend Vorkenntnisse, um sich selbst in dem Referenzmanual [LIN 91] zurechtzufinden. Als wahre Fundgrube erweist sich auch immer wieder die Newsgroup "comp.windows.interviews", in der InterViews-User aus der ganzen Welt Probleme und Tips austauschen.

Ein gutes Beispiel für die enormen Möglichkeiten, die unter InterViews programmierte Anwendungen bieten können, stellt die Diplomarbeit "Eine Experimentierumgebung für Mehrgitterverfahren" von Michael Arbesmeier [ARB 92] dar. Aufbauend auf InterViews und Unidraw, einer auf InterViews basierenden Bibliothek für graphische Editoren, wurde dort ein Programm geschaffen, das die interaktive Eingabe und Ausführung sogenannter Mehrgitteralgorithmen erlaubt. Weitere Informationen zu Mehrgitteralgorithmen und zu Unidraw sind in dieser Arbeit zu finden.

Abschließend möchte ich darauf hinweisen, daß mein Programmierstil nur einen Vorschlag darstellen kann. Die InterViews-Klassen wurden hier so verwendet, wie ich es aufgrund meines bisherigen Wissens für sinnvoll halte. Vielleicht haben die Väter von InterViews in einigen Punkten andere Vorstellungen als ich, nur sollten Sie sich dann einmal die Mühe machen, diese Richtlinien *schriftlich und in allgemein verständlicher Form* festzulegen. Ich hoffe jedenfalls, daß Ihnen dieses Büchlein ein bißchen Spaß bereitet hat und Sie hin und wieder eine kleine Anregung daraus entnehmen können.

Literaturverzeichnis

[ARB 92] Michael Arbesmeier, "Eine Experimentierumgebung für Mehrgittermethoden". Institut für Informatik, Technische Universität München, 1992.

[KNU 86] Donald E. Knuth, "Computers and Typesetting Vol. A: The TEX-book". Addison–Wesley Co., Inc., Reading, MA, 1986.

[LIN 91] Mark A. Linton et al., "InterViews Reference Manual Version 3.0". Board of Trustees of the Leland Stanford Junior University, 1991.

[PRO 87] Walter E. Proebster, "Peripherie von Informationssystemen". Springer–Verlag, Berlin Heidelberg, 1987.